uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería Telemática
2016/2017

Trabajo Fin de Grado

Desarrollo de un plugin para visualización de tráfico OpenID Connect

---

Rasha Aljelani Shellick

Tutores
Patricia Arias Cabarcos
Florina Almenares Mendoza

Leganés, 6 de julio de 2017

Agradecimientos:
Quiero aprovechar estas líneas para agradecer a todas las personas que me han ayudado y me han apoyado a lo largo de estos años en la Universidad Carlos III de Madrid.

En primer lugar quería agradecer el apoyo recibido por parte de toda mi familia, pasando por mis padres, Aljelani y Muheeba, y a mis hermanos, Rania, Mohamed y Lujain sin los cuales no habría acabado la carrera.

También quería agradecerles a mis tutoras, Patricia Arias Cabarcos y Florina Almenares Mendoza que me han ofrecido la posibilidad de trabajar en este proyecto tan interesante y que me ha resultado gratificante en cada momento recorrido.

Gracias a todos ellos.

Índice

Índice de figuras

Índice de tablas

1. Introducción

En esta sección se plantea una visión global del proyecto, con las respectivas motivaciones que han impulsado el trabajo y los objetivos que se pretenden conseguir y satisfacer.

1.1. Motivaciones

Esta era donde todo está informatizado, para facilitar el día a día, obliga a que la información sensible quede expuesta en cualquier ámbito de la red al tener que auntenticar en multitud de aplicaciones web.
Este problema hace aflorar nuevos protocolos de autenticación y autorización que permiten la centralización de los datos para una mayor seguridad, evitando así que se tenga la información digital replicada en distintas aplicaciones que podrían no ser todo lo seguras que se desea o espera.
Por esta razón se ha decidido la realización de un plugin en Wireshark, que es un programa que nos permite capturar el tráfico y cuya utilización está muy extendida entre los estudiantes y profesionales de las redes.
Gracias a la implementación del plugin los alumnos podrán analizar el tráfico transmitido a la hora de autenticar un usuario utilizando el protocolo OpenID Connect, siendo uno de los protocolos de gestión de identidad que facilita la unificación de la información para que sea posible el acceso a varias aplicaciones, puesto que se ha desarrollado para promover los siguientes puntos:

1. Ofrece una ayuda para el estudio del protocolo OpenID Connect, ya que además de identificar las trazas relacionadas con este protocolo, ofrece información relevante al respecto que refuerza el conocimiento teórico gracias a que sigue la guía de implementación paso a paso para los distintos flujos existentes. Por esta razón se usa como herramienta educativa para que los alumnos estudien OpenID Connect y puedan comprender la gran ayuda que representan estos protocolos en el día a día.

2. Da un uso práctico a la hora de la implementación de un cliente OpenID Connect, puesto que al analizar y mostrar los campos importantes de éste, permite una mayor depuración de las trazas enviadas por el cliente y la respuesta por parte del servidor, facilitando la detección y posterior corrección de errores.

1.2. Objetivos

El objetivo de este proyecto es desarrollar un plugin para analizar las trazas del protocolo OpenID Connect en el programa Wireshark y poder ofrecer soporte didáctico a los alumnos que estén estudiando los protocolos de gestión de identidad. Para ello se deben de cumplir los siguientes objetivos específicos:

1. Mostrar como se puede extender Wireshark a la hora desarrollar un analizador que sea capaz de, primero, identificar que una traza sea del protocolo a analizar, y segundo, catalogar cada una de ellas por las especificaciones del proveedor para poder comprender mejor los flujos.

2. Diseñar un programa en LUA, que es el lenguaje de programación escogido para el desarrollo de nuestro plugin, para que nuestro desarrollo ofrezca una vista de los puntos relevantes del protocolo OpenID Connect.

3. Exponer el funcionamiento de la infraestructura basada en cliente y servidor de un protocolo de gestión de identidad, para que se comuniquen entre ellos y capturar el tráfico generado entre ambas partes con el propósito de desarrollar el plugin y realizar las pruebas oportunas para su mejora.

1.3. Estructura de la memoria

La estructura de la memoria tiene seis grandes bloques diferenciados:

1. **Estado del arte**
   En este apartado se tratan las distintas tecnologías y protocolos que se han utilizado para el desarrollo del proyecto.

   - Infraestructuras de gestión de identidad:
     En esta sección se habla de las infraestructuras de gestión de identidad, describiendo su funcionamiento genérico para entrar en detalles sobre los principales protocolos que se han tenido en cuenta y las infraestructuras utilizadas por cada uno de ellos, Security Assertion Markup Language(SAML) y OpenID Connect, centrándonos en este último para tener una mayor base a la hora de comprender el funcionamiento del plugin y de las decisiones tomadas al desarrollarlo.
   - Servicios WEB:
     En este apartado se tratan las distintas tecnologías utilizadas para la comprensión de las peticiones enviadas con el protocolo OpenID Connect y el contenido que tendrán dichos paquetes. Por esta razón se describe Representational State Transfer (REST), JavaScript Object Notation (JSON), y JSON Web Tokens (JWT) que son servicios web en los que se apoya OpenID Connect para su funcionamiento.
   - Tecnologías:
     En este apartado se habla de las distintas tecnologías que se han utilizado a la hora de implementar y depurar el plugin. Wireshark, utilizado para analizar las trazas enviadas entre el cliente y el servidor, y OpenAM, una herramienta que nos ha permitido desplegar un servidor OpenID Connect junto a un cliente para realizar la autenticación.
     También se trata LUA, el lenguaje de programación utilizado para implementar el plugin, y de extensiones actualmente existentes para el protocolo de gestión de identidad SAML.
   - Marco regulador y entorno socioeconómico:
     Se describe también la normativa vigente hoy en día que se tiene que tener en cuenta para la realización del plugin y el desarrollo de la infraestructura creada para las pruebas.

2. **Desarrollo del plugin**
   En este bloque se habla más en profundidad sobre LUA y las principales

funciones para la creación de un disector, que es el nombre al que se da a los analizadores de Wireshark. También se desarrollan las medidas que se han tomado para ejecutar el plugin según las especificaciones de OpenID Connect.

3. **Arquitectura de despliegue**
   En este apartado se especifica el escenario final instalado y la configuración del cliente final. Se desarrolla el apartado de OpenAM en la parte de Tecnologías, explicando la instalación y configuración del servidor para que sea posible la autenticación de los usuarios y la comunicación con el cliente. Además se realiza un seguimiento de las trazas entre la parte cliente y servidor y se utiliza la explicación de OpenID Connect del apartado de Infraestructuras de gestión de identidad para exponer su funcionamiento real en un entorno de pruebas.

4. **Pruebas**
   En este apartado se detallan las pruebas realizadas sobre el plugin para dos bloques de pruebas:

   - Escenario local: realizar las pruebas utilizando un mismo equipo como servidor y cliente.
   - Escenario Distribuido: realizar las pruebas utilizando un equipo como servidor, y después otro distinto para la parte de cliente.

5. **Planificación y presupuesto**
   En este bloque se trata el tema de la planificación del proyecto y el presupuesto necesario para su ejecución.

6. **Conclusiones y líneas futuras**
   Se expone en este bloque las conclusiones a las que se han llegado desarrollando el plugin de OpenID Connect, cuáles son las mejores formas de optimización a largo plazo y cuáles son las lineas futuras de mejora.

2. Estado del arte

2.1. Infraestructuras de gestión de identidad

En este apartado se describe las infraestructuras de gestión de identidad de forma genérica para que una vez se tenga el conocimiento sobre sus principales características, entrar en profundidad sobre los dos principales protocolos que lo representan: SAML y OpenID.

1. SAML es el más antiguo de los dos y hoy en día es un estándar de protocolos de gestión de identidad, por lo que actualmente cuenta con multitud de plugin que analizan sus trazas en tiempo real.

2. El otro protocolo significativo es OpenID Connect, pero al tratarse de un estándar más nuevo y menos extendido entre las empresas y los desarrolladores, por lo que se mantiene en un segundo plano.

Lo primero que se debe comprender sobre este proyecto, serían las características de la infraestructura de gestión de identidad de forma genérica para comprender como funciona el plugin que se ha desarrollado y las trazas analizadas.

- **Definición de gestión de identidad**
  Antes de empezar describiendo la infraestructura de gestión de identidad, primero hay que responder a una pregunta ¿Que la gestión de identidad?
  Se denomina un sistema de gestión de identidad a un conjunto de políticas, tecnologías, infraestructura y procesos organizativos cuyo objetivo final es de gestionar y controlar el acceso de los usuarios a los sistemas de información[1].

- **Definiciones importantes**
  Una parte importante a tener en cuenta a la hora de comprender este tipo de infraestructuras es la diferencia entre estos dos conceptos:

  - **Autenticación:**
    A la hora de acceder a una aplicación que tiene restricciones en el acceso, se llama autenticación al proceso de verificación de la identidad del cliente, es decir, si alguien realmente es quien dice ser[2].
    Los mecanismos de autenticación existentes hoy en día son variados, pero el uso de usuario único y contraseña asociada es el más extendido.
  - **Autorización:**
    Una vez realizada la parte de autenticación del usuario, se realiza el proceso de autorización. Se llama autorización al proceso de establecer reglas que determinan los permisos del usuario que accede al sistema. es decir: ¿a quién se le permite hacer qué?
    Los protocolos de gestión de identidad permiten que un usuario autorice el acceso a sus datos personales en una aplicación a otra tercera parte externa que necesita esa información.

- **Roles de la infraestructura de gestión de identidad**
  La gestión de identidad dispone de diferentes roles en la infraestructura básica de cualquier protocolo.

El objeto de estos roles es separar funcionalidades y por ello conseguir una mayor seguridad a la hora de intercambiar información entre los sistemas de información y usuarios.A continuación se describen estos roles:

- **Usuario final**:
  Usuario que quiere acceder a una aplicación tercera utilizando sus credenciales que mantiene en un sistema de información y con los cuales intenta autenticar. Puede realizar la autenticación a través de un agente de usuario como el navegador, dispositivo móvil, escritorio virtual, o similar.
- **Proveedor de identidad (*Identity Provider* (IdP)):**
  Es la entidad encargada de emitir la información relacionada con el usuario a los proveedores de servicios que necesiten confirmar la identidad del mismo.
  También es el proveedores que posee la infraestructura de autenticación e implementa las funciones de gestión de los usuarios.
- **Proveedor de servicios (*Service Provider* (SP)):**
  Es la entidad que da acceso al usuario o a un sistema de información tercero qué confían en el *IdP* para llevar a cabo las tareas de autenticación y acceso al recurso que está solicitando.

Cada uno de los roles tiene una función específica que llevada en conjunto brindan la seguridad en las infraestructuras de gestión de identidad e interactúan para garantizar una multitud de funcionalidades importantes a la hora de garantizar el acceso a la información confidencial del usuario.

- **Funcionalidades de gestión de identidad**
  Las funcionalidades que tienen este tipo de infraestructuras son bastante amplias, pero este apartado se centra en las principales que existen hoy en día[3]:

  - **Control de acceso:** consiste en la verificación de lo privilegios a determinado usuarios o entidades para acceder a algún recurso protegido.
  - **Gestión de identidad digital:** es la gestión de la información y la identidad digital de un usuario en determinadas aplicaciones.
  - **Gestión de contraseñas:** consiste en la capacidad de almacenaje, recuperación y generación de contraseñas en una aplicación a partir de una base de datos cifrada y segura, facilitando a un usuario o entidad auto-gestionar su propia contraseña.
  - **Automatización de flujos de trabajo:** mediante mecanismos de intercambio de información se consigue automatizar los flujos de trabajo para la organización de recursos en distintos procesos.
  - **Single sign-on:** consiste en el procedimiento de autenticación que permite a los usuarios finales o la administración de las entidades al acceso a varios sistemas mediante un inicio de sesión único.

- **Servicio de *Token* de Seguridad:** es el mecanismo de control de acceso basado en emitir testigos (*Tokens*) de seguridad para el intercambio de información sensible del usuario final.
- **Control de acceso en base a roles:** consiste en la asignación de roles a distintos usuarios para después definir políticas de seguridad basada en autenticación e identificación de los distintos roles.

Con estas funcionalidades, y otras menos relevantes y que no se han descrito en este apartado, las infraestructuras de gestión de identidad consiguen administrar el acceso de los usuarios finales.

Habiendo descrito las principales característica de forma genérica de la infraestructura de gestión de identidad, ahora se detallan los protocolos más utilizados a día de hoy en gestión de identidad son OpenID Connect y SAML.

2.1.1. OpenID Connect

Como ya se ha mencionado con anterioridad en el apartado **2.1. Infraestructuras de gestión de identidad**, OpenID Connect es un protocolo de gestión de identidad de usuarios que nos permite conocer si un usuario ya está autenticado en una aplicación. En caso contrario habilita la autenticación mediante el uso de una credencial común permitiendo el acceso a distintas aplicaciones con una única cuenta de registro.
Esto nos permite el acceso a distintas aplicaciones con una única cuenta de registro, lo que ayuda a no tener distintas cuentas con los mismos datos provocando al final una ineficiente y poco segura forma de gestión de usuarios con contraseñas débiles[4].
Es un protocolo que va una capa por encima de Oauth 2.0[6], lo que facilita la integración tanto de la parte de la autenticación como de la autorización con un único paso y simplificado, al ser el envío de sus mensajes en base JSON, término explicado en el apartado **2.2.2. JSON**.
Se puede describir la estructura de los conjuntos de protocolos que hacen OpenID Connect funciona de la siguiente manera:

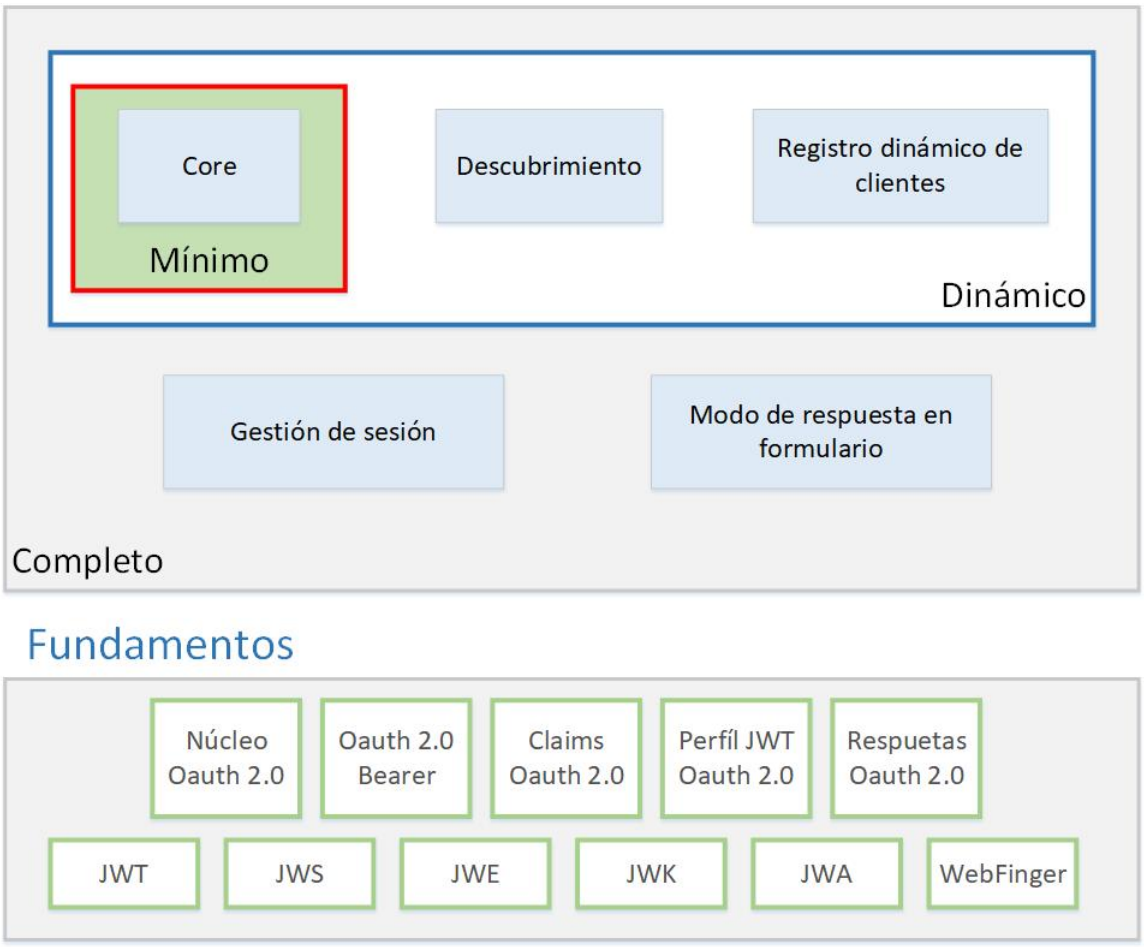

Figura 1: Conjunto de protocolos de OpenID Connect

- **Oauth 2.0**
  Oauth es un estándar de flujos de autorización que permite a terceros acceder a los contenidos propiedad de un usuario final alojado en una determinada aplicación, sin que tengan credenciales de autenticación.
  La primera implementación de Oauth no tuvo tanto éxito debido a la complejidad del protocolo y a que únicamente se podía utilizar en peticiones HTTP donde el usuario final se autenticaba a través de navegadores web, por lo que era imposible realizar la autorización mediante una aplicación móvil o de escritorio. Por esta razón se creó otra versión mejorada, Oauth 2.0, donde se suplían las carencias anteriormente mencionadas.
  Como ya se ha comentado, Oauth es un protocolo de autorización, pero no de autenticación, por lo que siempre necesita utilizar el mecanismo de autenticación del servidor web de destino o, como en el caso de OpenID Connect, una capa superior que realice el proceso de autenticación e intercambie las credenciales entre los distintos actores.
  Se puede explicar este hecho con un ejemplo simple para entender la diferencia:

Figura 2: Autenticación utilizando OpenID Connect

Figura 3: Pseudo autenticación utilizando Oauth

En la **Figura 2** que representa un proceso de autenticación utilizando OpenID Connect se observar que la respuesta del proveedor de identidad es una afirmación de identidad del usuario al autenticarse. Sin embargo, en la **Figura 3** se ve que, a diferencia de la autenticación usando OpenID Connect, no se envía directamente la afirmación de autenticación del usuario, sino que la aplicación envía un *Token* de acceso que puede conceder a la aplicación acceso al IdP en nombre del usuario y que pueden adjuntar a sus peticiones para demostrar que tienen permiso por parte del mismo.
En los siguientes apartados se explican los flujos de autenticación necesarios para que un usuario final acceda a una aplicación donde no tiene cuenta creada y se detallan mejor tanto el protocolo OpenID Connect como Oauth 2.0 y como interactúan ambos para realizar la tarea.

- **Principales roles**
  Antes de explicar el funcionamiento del protocolo habría que dar una pequeña introducción sobre los actores principales que entran en juego:

  - **Relying Party (RP)**
    En términos generales de infraestructura de gestión de identidad, este es el rol de IdP.
    Es el nombre que se le da al cliente de OpenID Connect, ya que una vez se realiza la autenticación, el servidor envía las reclamaciones (*claims*) confidenciales del usuario al cliente, de una forma segura, para que pueda confiar en él. Por esta razón se llama la parte que confía (*relying party*).

- **OpenID Provider (OP)**
  Es el servidor de autorización que autentica al usuario final y proporciona la información relacionada con la autenticación que se ha realizado y sobre el usuario final en forma de *claims* al *relying party*. El *OpenID Provider* lo componen varias partes:
  - Punto final de autorización (Authorization Endpoint): servidor que tiene la función de autorizar en el flujo del protocolo.
  - Punto final de Token (Token Endpoint): recurso protegido que devuelve el *Token* de Acceso y el *ID Token*.
  - Punto final de Userinfo (Userinfo Endpoint): recurso protegido que devuelve las *claims* sobre el usuario final autenticado.
- **Usuario final (End-user)**
  Usuario final que intenta acceder a una aplicación identificándose con una cuenta creada en un servidor que soporta el protocolo OpenID Connect, a través de una aplicación tercera.
- **Agente de usuario (User-agent)**
  Aplicación que actúa como cliente cuando se accede a la red, que en este caso y de la infraestructura a explicar, se trata de un navegador web o de un dispositivo móvil. En las cabeceras de las peticiones HTTP se identifica la aplicación que se ha usado, en el campo user-agent, donde se detalla el nombre, la versión de la aplicación, etc.

- **Flujo general de OpenID Connect**
  En términos generales se puede entender el flujo de OpenID Connect de la siguiente manera:

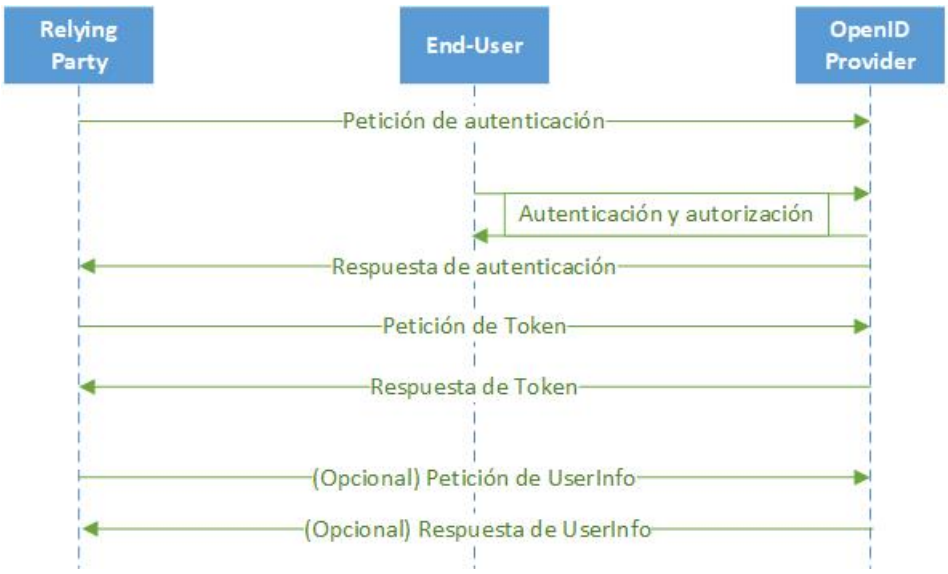

Figura 4: Flujo general de OpenID Connect

Como se puede observar en la **Figura 4**, y según las especificaciones de OpenID Connect, el flujo general sigue los siguientes pasos:

1. Primero el cliente, *relying party* (RP), envía una petición de autenticación al *OpenID Provider* (OP).
2. El OP autentica al usuario final, mediante la redirección del usuario a un menú de autenticación, y obtiene una autorización por su parte.
3. El OP realiza una petición de respuesta a la petición del RP inicial agregando un *ID Token* y opcionalmente un Token de Acceso si fuera necesario.
4. Una vez recibida el RP puede enviar un Petición de UserInfo agregando el Token de Acceso a la petición, para poder recibir más detalles del usuario.
5. El OP devuelve *claims* sobre el usuario final en respuesta de la última petición.

En OpenID todas las peticiones se realizan mediante el protocolo HTTP, por lo que se puede realizar un seguimiento de todas ellas desde el agente de usuario utilizado; ya que en base a peticiones de redirección y de envío se puede conseguir que el usuario final acabe autenticando en el cliente, usando sus credenciales de su proveedor de OpenID.

- **Principales definiciones de OpenID Connect**
  OpenID Connect tiene propiedades y estructuras que la hacen singular, y por ello en este apartado se exponen las principales definiciones que la caracterizan.

  - **Claim**
    En OpenID Connect existe el término de *claim*, que es la información confirmada por parte del servidor de OpenID Connect y que envía al cliente una vez se autentica el usuario. Nos sirve a la hora de autenticar el usuario, al autorizarlo o incluso en la fase inicial al enviar el Servidor de OpenID Connect información relevante sobre el perfil del usuario final, nombre, foto, etc.
    Este término se describe en los siguientes apartados para explicar mejor su funcionamiento en las distintas etapas de autenticación de OpenID Connect.
  - **ID Token**
    La principal característica que se ha implementado en OpenID Connect frente al otros protocolos de autenticación, como es el caso de Oauth 2.0, es que ha sabido simplificar sus peticiones mediante el uso de la estructura creada, *ID Token*.
    La extensión consiste en un *Token* de seguridad que contiene *claims* sobre la autenticación de un usuario final y que es enviada por un servidor de autorización cuando se utiliza un *relying party*, encapsulado en un JSON Web Token [JWT].
    Los principales *claims* que componen el *ID Token* son los siguientes:

Claim	requisito	Descripción
iss	Obligatorio	(Issuer Identifier) Un identificador para el Emisor de la respuesta. El valor iss es una dirección URL sensible a mayúsculas y minúsculas, utilizando HTTPS, que contiene el *schema, host* y opcionalmente, número de puerto y componentes de *path* y ninguna *query* o componentes de fragmentos.
sub	Obligatorio	(Subject Identifier) Un identificador único y nunca reasignado dentro del emisor para el Usuario Final, que está destinado a ser consumido por el cliente.
aud	Obligatorio	Audiencia a la que está destinado este *Token* de Identificación, y que debe contener el *client_id* de OAuth 2.0 del *relying party.*
exp	Obligatorio	Tiempo de expiración del *ID Token.* Para cumplir con el procesamiento de la petición se requiere que en el momento de resolver la petición de este parámetro, la fecha y hora actual DEBEN estar antes de la fecha y hora de caducidad indicada en el valor.
iat	Obligatorio	Momento en que se emitió el JWT. Su valor es un número JSON que representa el número de segundos desde 1970-01-01T0: 0: 0Z, medido en UTC hasta la fecha y hora.
auth_time	Obligatoria u opcional	Hora en que se produjo la autenticación del usuario final. Su valor es un número JSON que representa el número de segundos desde 1970-01-01T0: 0: 0Z, medido en UTC hasta la fecha y hora de ese momento. Cuando se realiza una solicitud de *max_age* o cuando se solicita *auth_time* de forma obligatoria se incluye, de lo contrario, su inclusión es opcional.
Continuación de la tabla		

Tabla 1 – Continuación de las principales características de *ID Token*.

Claim	requisito	Descripción
nonce		Valor de cadena utilizado para asociar una sesión del cliente con un *ID Token* y para mitigar los ataques por repetición. El valor no se modifica desde la petición de autenticación hasta el ID. Si está presente en el *ID Token*, los clientes deben verificar que el valor de nonce es igual al valor del parámetro nonce enviado en la solicitud de autenticación. Si están presentes en la Solicitud de Autenticación, los Servidores de Autorización deben incluir el *claim* en el *ID Token* con el valor de nonce, que es el valor que es enviado en la petición de autenticación.
acr	Opcional	(Authentication Context Class Reference) Cadena de caracteres que identifica la Autentication Context Class que realizó una autenticación satisfactoria.
amr	Opcional	(Authentication Methods References) Es un JSON compuesto por un conjunto de cadenas de texto que representan identificadores para los métodos de autorización que se pueden utilizar. Un ejemplo de este *claim* sería tener unos valores que indiquen que se han utilizado contraseña y OTP para la autenticación.
azp	Opcional	(Authorized party) La parte a la que se emitió el *ID Token*. Si está presente, debe contener el ID de cliente OAuth 2.0 de esta parte. Esta Reclamación sólo es necesaria cuando el *ID Token* tiene un valor de audiencia único y esa audiencia es diferente que la parte autorizada. Puede ser incluido incluso cuando la parte autorizada es la misma que la audiencia única.

Tabla 1: Principales características de *ID Token*.

- **Autenticación**
  Ahora que se tiene una visión global del flujo general de OpenID Connect y se

entienden mejor los actores que entran en juego y la principal característica que hace distinto este protocolo, se puede profundizar sobre la autenticación. La autenticación puede seguir distintos flujos, según las especificaciones de OpenID Connect, dependiendo de cómo se vaya a realizar la entrega del *ID Token* y de *Token* de Acceso al cliente. Los flujos existentes son los siguientes:

- Flujo Código (Authentication Code Flow):
  Se caracteriza porque el *claim response_type* es igual a *code*.
- Flujo Implícito (Authentication Implicit Flow):
  se caracteriza porque el *claim response_type* es igual a *id_token token* o *id_token*.
- Flujo Híbrido (Authentication Hybrid Flow):
  Utiliza otros valores de *response_type* definidas en OAuth 2.0.

La **tabla 2** muestra un resumen donde se puede observar el mejor flujo a considerar y que se amolda a la infraestructura:

Propiedades	**Flujo Código**	**Flujo Implícito**	**Flujo Híbrido**
Todos los *Tokens* son devueltos por el autorizador final	no	Si	No
Todos los *Tokens* son devueltos por el *Token* Endpoint	Si	No	No
Los *Tokens* no son relevantes para el agente de usuario	Si	No	No
El cliente puede ser autenticado	Si	No	Si
Es posible utilizar un Refresh *Token*	Si	No	Si
Comunicación en una única ronda	No	Si	No
Comunicación de servidor a servidor	Si	No	Varía

Tabla 2: Características de los flujos OpenID Connect

Como se puede observar por la **tabla 2** se escoge un flujo frente a otro dependiendo de las características de conexión buscadas y de los mecanismos de autenticación utilizados.
Ahora se explica cada uno de los tipos de flujos cuyo resultado común es que una vez realizada la autenticación mediante el protocolo OpenID Connect se devuelve en un *ID Token*, con toda la información que se ha explicado con anterioridad.

- **Autenticación mediante Flujo Código**
  Cuando se utiliza esta forma de autenticación el servidor devuelve devuelve un código de autorización para el cliente que puede cambiar por un *ID Token* y un *Token* de Acceso, y todos ellos son devueltos desde el *Token Endpoint*.
  El Flujo Código ofrece la ventaja de no exponer ningún *Token* al

agente de usuario y posiblemente a otras aplicaciones maliciosas con accceso al agente de usuario. El Servidor de Autorización puede también autenticar al cliente antes cambiando el código de autorización para conseguir un *Token* de Acceso.
Este tipo es adecuado para clientes que pueden mantener de forma segura una clave secreta ente ellos y el servidor de autorización.

Pasos del Flujo Código: La **Figura 5** muestra el proceso de autenticación mediante flujo de código.

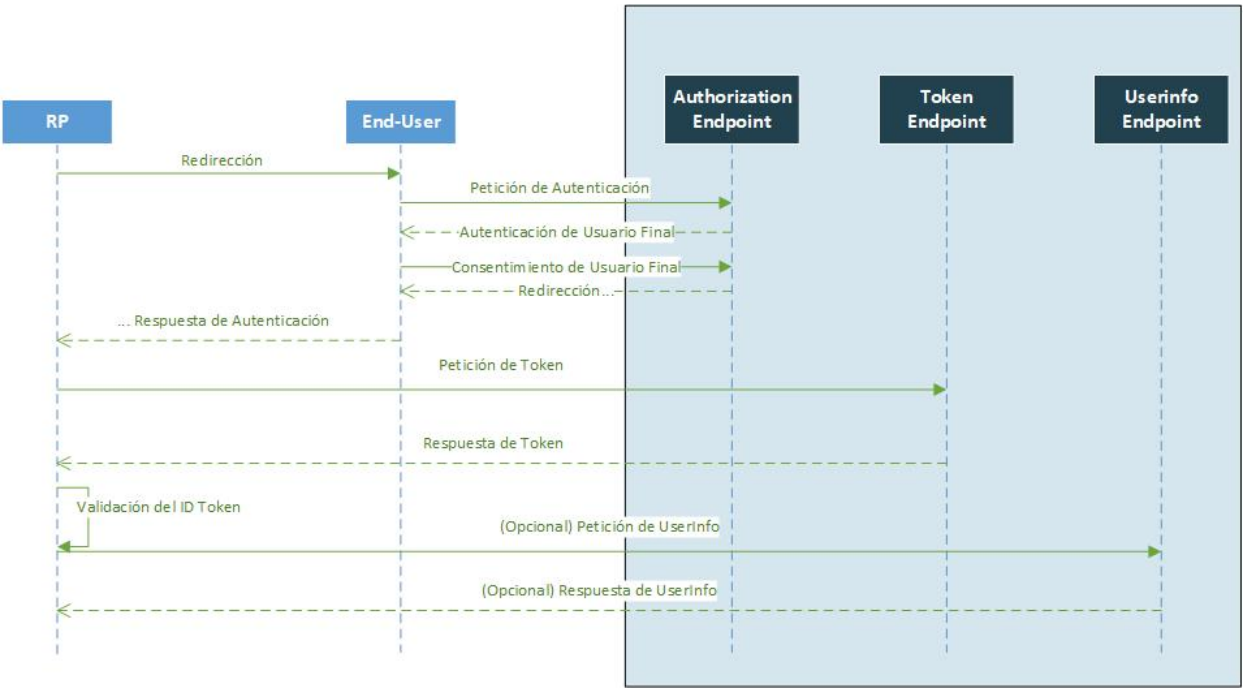

Figura 5: Flujo de Código

1. *Petición de autenticación (Authentication Request)*:
   Es una petición que solicita que el usuario final esté autenticado por el servidor de autenticación. Estos servidores deben de soportar las peticiones GET y POST de HTTP, ya que los clientes usan estos métodos para enviar la petición adecuadamente al servidor.
   Para que el agente de usuario realice la petición de autenticación, el cliente fuerza que se haga mediante un redireccionamiento que lo activa:

```
HTTP/1.1 302 Found
Location: https://server.example.com/authorize?
response_type=code
&scope=openid%20profile%20email
&client_id=s6BhdRkqt3
&state=af0ifjsldkj
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
Host: server.example.com
```

Si se utilizan la peticiones GET, los parámetros de la petición se serializan con *URI Query String Serialization* , mientras que si son peticiones POST se serializan con *Form Serialization*. Estos serían algunos ejemplos expuestos en las especificaciones de OpenID Connect:

```
GET /authorize?
response_type=code
&scope=openid
&client_id=s6BhdRkqt3
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
HTTP/1.1
Host: server.example.com
```

```
POST /authorize HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

response_type=code
&scope=openid
&client_id=s6BhdRkqt3
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
```

En las peticiones de OpenID se utiliza tanto parámetros de este protocolo como de parámetros de OAuth 2.0, pero los principales *claims* que se tienen que presentar en el protocolo del que estamos haciendo referencia son los siguientes:

Claim	**requisito**	**Descripción**
scope	Obligatorio	Si no se tiene este valor de *openid* en la petición, el comportamiento está sin especificar.
		Continuación de la tabla

Tabla 3 – Continuación de las Principales *claims*

Claim	requisito	Descripción
response_type	Obligatorio	Es el valor que determina el flujo de procesamiento de la autorización que es utilizado, incluyendo qué parámetros son devueltos por los usuarios finales. Cuando se utiliza el Flujo Código, este valor es code.
client_id	Obligatorio	Identificador del cliente OAuth 2.0 valido en el Servidor de Autorización.
redirect_uri	Obligatorio Redirección	URI a la que se envía la respuesta. Este URI debe coincidir exactamente con el valor de URI de redirección para el cliente pre-registrado en el proveedor de OpenID.
state	Recomendado	Valor que se utiliza para mantener el estado entre la solicitud y el *callback*. Típicamente, la mitigación Cross-Site Request Forgery (CSRF, XSRF) se realiza mediante la unión criptográfica de este parámetro con una cookie del navegador.
Continuación de la tabla		

Tabla 3 – Continuación de las Principales *claims*

Claim	requisito	Descripción
prompt	opcional	Lista de cadenas de caracteres ASCII delimitada por espacio, que especifica si el servidor de autorización solicita al usuario final como autenticar: **none:** el servidor de autorización no debe de mostrar ninguna página de autenticación para el usuario, devolviendo error del tipo *login_required*, *interaction_required* u otro código de los descritos en la [RFC6749] en el caso de que no esté autenticado. Igualmente más adelante, en la sección de respuesta de autenticación describimos algunos de los mensajes de error. **login:** el servidor de autorización debe de solicitar al usuario final que vuelva a autenticar, y en caso de que no se pueda devuelve un error de *login_required*. **consent:** el servidor de autorización debe de solicitar al usuario final su consentimiento para enviar la información al cliente, y en el caso de que no se consiga devuelve un error del tipo *consent_required*. **select_account:** el servidor de autorización debe de solicitar al usuario final que seleccione una cuenta de usuario, ya que tiene varias creadas en el mismo servidor. En caso de que no seleccione ninguna cuenta devuelve un error del tipo *account_selection_required*.
nonce	opcional	Cadena de caracteres utilizada para asociar una sesión de Cliente con un *ID Token* y para mitigar los ataques de repetición.

Continuación de la tabla

Tabla 3 – Continuación de las Principales *claims*

Claim	requisito	Descripción
azp	opcional	(Authorized party) La parte a la que se emitió el *ID Token*. Si está presente, debe contener el ID de cliente OAuth 2.0 de esta parte. Esta Reclamación sólo es necesaria cuando el *ID Token* tiene un valor de audiencia único y esa audiencia es diferente que la parte autorizada.
max_age	Optional	(Maximum Authentication Age) Indica el tiempo máximo permitido en segundos desde la última vez que el usuario final fue autenticado por el Proveedor de OpenID.

Tabla 3: Principales *claims* de la petición de autenticación

El servidor de autorización debe validar la solicitud recibida de la siguiente manera:

a) Primero el servidor de autorización debe validar todos los parámetros de OAuth 2.0 de acuerdo con la especificación de OAuth 2.0.

b) También tiene que comprobar que el valor del *claim* scope sea igual a *openid* para tratarse del protocolo OpenID Connect. Si scope no tiene ese valor, puede aún así tratarse del protocolo OpenID Connect pero no sería una petición. En los ejemplos que se han puesto con anterioridad, se puede observar que tanto para las peticiones POST como GET tienen el *claim* scope=openid.

c) El servidor de autorización debe comprobar que todos los parámetros obligatorios se encuentran y se ajustan a la petición. En los ejemplos anteriores se pueden observar que todos los *claims* obligatorios (*response_type*, *scope*, *client_id* y redirrect_uri) aparecer en la petición.

2. *El servidor de autorización autentica al usuario final*:
Si la petición de autenticación es válida, el servidor de autorización autentica al usuario final o determina si se mantiene autenticado, dependiendo de los valores de los parámetros utilizados en la solicitud. Por ello en la petición tiene que tener el parámetro *prompt* el valor de login.
Para que pueda autenticar, se le redirige a otra dirección donde

le muestra al usuario final un menú en el cual puede introducir su usuario y contraseña, en el caso de que se haya utilizado ese método de autenticación, mediante una cookie de sesión, etc.
El servidor de autenticación siempre autentica al usuario si este no se encontraba autenticado con anterioridad, o si el valor del *claim* prompt es igual a login. En este caso se le pide que autentique aún cuando el usuario ya estuviera de por si autenticado.
Por el contrario, el servidor de autenticación no realiza ninguna interacción con el usuario final si el valor prompt es igual a none devolviendo una respuesta de error.

3. *El servidor de autorización obtiene el consentimiento del usuario final*:
Una vez esté autenticado el usuario final, el servidor de autorización debe obtener la autorización por parte del usuario antes de enviar ningún tipo de información. Esto se realiza a través de un diálogo interactivo con el usuario final en el que aparecen los parámetros a los cuales está consistiendo. Un ejemplo del diálogo de autorización podría ser el que pide Google Oauth 2.0 al utilizarlo como servidor de autorización:

Figura 6: Menú de despliegue para pedir consentimiento

4. *Respuesta de autenticación (Authentication Response)*:
En una petición de autenticación correcta, se envía una respuesta de autenticación satisfactoria del servidor de autenticación al cliente utilizando la URL que aparece en response_uri concatenando los campos obligatorios que vienen definidos en la sección 4.1.2 de OAuth 2.0 [RFC6749], es decir, añadiendo obligatoriamente el campo code y que tiene el valor del código de autorización usado posteriormente para obtener el *ID Token* y el *Token* de Acceso. También tiene que cumplir otra característica, y es que el formato tiene que ser de tipo application/x-www-form-urlencoded.
Una respuesta correcta en este caso podría ser el siguiente:

```
HTTP/1.1 302 Found Location: https://client.example.org/cb?
code=SplxlOBeZQQYbYS6WxSbIA
&state=af0ifjsldkj
```

Una vez conseguida una respuesta de autenticación correcta, únicamente falta que el cliente valide todos los campos de la respuesta acorde a la [RFC6749]. Este paso en el Flujo Código es obligatorio.
Por otro lado, para los parámetros que aparecen en la petición de autenticación y que no se especifican ni son reconocidos en el protocolo de OAuth 2.0, deberán ser ignorados por el servidor de autorización y si la autenticación falla de algún modo, debe devolver una respuesta de error.
Los *claims* en una respuesta de autenticación errónea son los siguientes:

Claim	requisito	Descripción
error	Obligatorio	Código de error.
error_description	Opcional	Descripción del error en decodificado en caracteres ASCII.
error_uri	Opcional	URI de la página web donde se describen de forma adicional los detalles del error.
state	Obligatorio si está incluido el parámetro state en la petición de autorización.	Es el valor del parámetro recibido por el cliente.

Tabla 4: *Claims* obligatorios de la respuesta de autenticación fallida

El *claim* de error puede tener los siguientes valores según las especificaciones de OpenID Connect:

- login_required: el servidor de autenticación requiere que el usuario final esté autenticado. Este error se devuelve cuando el valor de *prompt* es igual a *none* pero la petición no se ha podido completar porque no se ha podido desplegar la interfaz de autenticación para el usuario.
- account_selection_required: El usuario final es requerido para seleccionar una sesión del servidor de autenticación, ya que puede ser autenticado en el servidor de autorización con diferentes cuentas asociadas, pero el usuario final no seleccionó una sesión.
- consent_required: el servidor de autenticación requiere el

consentimiento del usuario final.
- invalid_request_uri:el parámetro *request_uri* de la petición de autenticación devuelve un error o contiene datos inválidos.
- invalid_request_object:el parámetro *request* contiene un objeto *request* inválido.
- request_not_supported: el proveedor de OpenID Connect no soporta el parámetro *request*.
- request_uri_not_supported: el proveedor de OpenID Connect no soporta el parámetro *request_uri*.
- registration_not_supported: el proveedor de OpenID Connect no soporta el parámetro *registration*.

Un ejemplo de respuesta de error que se especifica en la página de OpenID Connect sería el siguiente:

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb?
error=invalid_request
&error_description= Unsupported %20response_type %20value
&state=af0ifjsldkj
```

5. *Petición del* Token *(Token Request)*:
Para obtener un *Token* de Acceso, un *ID Token*, y opcionalmente un *Refresh Token*, el cliente envía una petición del *Token* al *Token Endpoint* para obtener una respuesta del Token siempre mediante TLS.
Un cliente realiza un Petición de *Token* mediante la presentación de su código de autorización, obtenido previamente en la respuesta de autenticación tal como se ha explicado en el punto cuatro, al *Token Endpoint* utilizando el valor authorization_code = grant_type. Si el cliente es un cliente confidencial, entonces se deberá autenticarse en el *Token Endpoint* mediante el método de autenticación registrado para su *client_id*. El cliente envía los parámetros al Token Endopoint utilizando el método POST del protocolo HTTP, con Form Serialization. Un ejemplo de una posible petición de Token descrita en las especificaciones de OpenID Connect sería la siguiente:

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded
Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW

grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
&redirect_uri=https %3A %2F %2Fclient.example.org %2Fcb
```

Para que el servidor de autenticación pueda validar que estos parámetros son los correctos, se realizan las siguientes comprobaciones:

a) Primero habría que asegurarse de que el código de autorización se entrega al cliente autenticado y no a otro usuario.

b) Segundo se comprueba que el código de autorización es válido.

c) Si es posible, lo ideal sería verificar que el código de autorización no ha sido utilizado anteriormente.

d) Asegúrese de que el valor del parámetro *redirect_uri* en el Petición de *Token* es idéntico al que se incluyó en la solicitud inicial de autorización.

6. *Respuesta del Token (Token Response)*:
   Después de recibir y validar una Petición de *Token* válida y autorizada del cliente, el servidor de autorización devuelve una respuesta exitosa que incluye un *ID Token* y un Token de Acceso. La respuesta utiliza el tipo de contenido application/JSON, devolviendo todos los valores del Token en un JSON cifrado.
   El valor del parámetro *token_type* de la respuesta debe ser igual a *Bearer*, especificado en la RFC OAuth 2.0 Bearer Token Usage [RFC6750], a menos que otro Token Type se haya negociado con el cliente. En cualquier caso los servidores deben de soportar que el valor del parámetro *token_type* sea igual a Bearer.
   El único parámetro obligatorio en OpenID Connect en las Respuesta de *Token*, aparte de los especificados en el protocolo Oauth 2.0, es el *claim id_token* que tiene el valor del *ID Token* asociado a la sesión autenticada.
   Al tratarse de una petición HTTP que contiene datos sensibles, como son los *Tokens*, siempre tiene que tener estas dos cabeceras de respuesta con los dos siguientes valores:
   - Cache_control: no-store
   - Pragma: no-cache

   Una posible respuesta exitosa del protocolo OpenID Connect podría ser el siguiente ejemplo que se ha encontrado en las especificaciones de OpenID Connect:

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
”access_token”: ”SlAV32hkKG”,
”token_type”: ”Bearer”,
”refresh_token”: ”8xLOxBtZp8”,
”expires_in”: 3600,
”id_token”: ”eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ.ewogI
mlzc yI6ICJodHRwOi8vc2VydmVyLmV4YW1wbGUuY29tIiwKICJ
zdWIiOiAiMjQ4Mjg5NzYxMDAxIiwKICJhdWQiOiAiczZCaGRSa3
F0MyIsCiAibm9uY2UiOiAibi0wUzZfV3pBMk1qIiwKICJleHAiOi
AxMzExMjgxOTcwLAogImlhdCI6IDEzMTEyODA5NzAKfQ.ggW8
hZ1 EuVLuxNuuIJKX_V8a_OMXzR0EHR9R6jgdqrOOF4daGU96Sr
_P6qJp6IcmD3HP99Obi1PRs-cwh3LO-
p146waJ8IhehcwL7F09JdijmBqkvPeB2T9CJNqeGpe-
gccMg4vfKjkM8FcGvnzZUN4_KSP0aAp1tOJ1zZwgjxqGByKHiO
tX7TpdQyHE5lcMiKPXfEIQILVq0pc_E2DzL7emopWoaoZTF
_m0_N0YzFC6g6EJbOEoRoSK5hoDalrcvRYLSrQAZZKfly
uVCyixEoV9GfNQC3_osjzw2PAithfubEEBLuVVk4XUVrWO
LrLl0nx7RkKU8NXNHq-rvKMzqg”
}

Como se puede ver en la traza, cumple con los valores de las cabeceras obligatorias por tener información sensible, Cache-Control y Pragma. También se puede observar que el valor de *id_token* muestra el valor del *ID Token* codificado y que al descodificarlo tendrá el siguiente valor:

{"alg":"RS256","kid":"1e9gdk7"}

{
"iss": "http://server.example.com",
"sub": "248289761001",
"aud": "s6BhdRkqt3",
"nonce": "n-0S6_WzA2Mj",
"exp": 1311281970,
"iat": 1311280970
}
ggW8hZ1EuVLuxNuuIJKX_V8a_OMXzR0EHR9R6jgdqrOOF4
daGU96Sr_P6qJp6IcmD3HP99Obi1PRs-cwh3LO-p146wa
J8IhehcwL7F09JdijmBqkvPeB2T9CJNqeGpe-
gccMg4vfKjkM8FcGvnzZUN4
_KSP0aAp1tOJ1zZwgjxqGByKHiOtX7TpdQyHE5lcMiKPXf
EIQILVq0pc_E2DzL7emopWoaoZTF_m0_N0YzFC6g6EJbOEo
RoSK5hoDalrcvRYLSrQAZZKflyuVCyixEoV9GfNQC3_osjzw2PA
ithfubEEBLuVVk4XUVrWOLrLl0nx7RkKU8NXNHq-rvKMzqg"

El valor del parámetro *id_token* es el *ID Token*, que tiene formato JWT, contiene tres segmentos codificados en base64url separados por los caracteres de punto ('.'). El primer segmento representa la cabecera JOSE, el segundo segmento representa los *claims* del *ID Token* y el tercero la firma.
Más adelante, en la sección **2.2.3. JWT** se habla de JWT y como se descifra la información que contiene.

En cambio, en el caso de que la petición no es válida o no está autorizada, devuelve un error como está definido en la sección 5.2 de OAuth 2.0 [RFC6749]. Al igual que en el caso afirmativo, el cuerpo del mensaje HTTP será un JSON con el código de respuesta 400:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json
Cache-Control: no-store
Pragma: no-cache

{
"error": "invalid_request"
}
```

Para validar el Respuesta de *Token*, hay que tener en cuenta los siguientes factores:

a) Tiene que seguirlas validaciones que se describen en la [RFC6749] de Oauth 2.0.

b) Tiene que validar el *ID Token*:
Antes que nada, cabe aclarar que aparte de los *claims* mencio-

nados anteriormente al hablar del *ID Token*, hay uno extra que únicamente entra en juego en el Flujo Código:

Claim	requisito	Descripción
at_hash	Opcional	Su valor es la codificación de base64url de la mitad izquierda del hash de los octetos en ASCII del valor access_token, donde el algoritmo de hash utilizado es el algoritmo hash utilizado en el parámetro alg expuesto en el *ID Token*.

Tabla 5: *ID Token claims* en Flujo Código

Los pasos para validar el *ID Token* son los siguientes:

1) Si el *ID Token* está cifrado, habría que descifrarlo utilizando la clave y el algoritmo que el cliente especificó durante el registro, ya que es como el servidor ha cifrado el *ID Token*. En cambio, si fue negociado que fuera cifrado pero en cambio el *ID Token* no lo está, entonces el cliente tiene que rechazarlo.
2) El identificador del emisor tiene que coincidir con el valor de *iss* en el *ID Token*.
3) Tiene que comprobar que el valor del *claim* aud tiene el mismo valor que el parámetro *client_id* en la petición de autenticación.
4) Si el *ID Token* contiene múltiples audiencias, tiene que comprobar que el parámetro *azp* está presente.
5) Si el valor azp está presente, el cliente debe comprobar que el valor de *client_id* es igual al valor de ese claim.
6) El Cliente debe validar la firma de todos los demás *ID Tokens* de acuerdo con JWS usando el algoritmo especificado en el parámetro de cabecera de algoritmo de JWT, alg. En el caso del Flujo Código, la validación del servidor TLS puede ser usada para validar al emisor en vez de comprobar la firma del *ID Token*.
7) El valor del *claim alg* puede ser el de por defecto, RS256, o por el contrario uno negociado previamente en el parámetro *id_token_signed_response_alg* durante el registro.
8) Si el parámetro de cabecera de algoritmo de JWT, alg, utiliza un algoritmo basado en MAC como es el caso de HS256, HS384 o HS512, se usan los octetos de la representación UTF-8 de la *client_secret* correspondiente al *cliente_id* contenido *aud* como clave para validar la firma.

9) El tiempo no puede ser posterior al tiempo que fija el parámetro *exp*.
10) Si el tiempo transcurrido está muy alejado del tiempo marcado por el parámetro *iat* se rechaza la petición.
11) Se debe de comprobar que el valor del *claim nonce* es el mismo que el valor del mismo *claim* en la petición de autenticación.
12) Si se pidió el valor de *acr* en la petición, se debe de comprobar que sea válido.
13) Si se ha solicitado el *claim auth_time*, ya sea a través de una solicitud específica para este *claim* o utilizando el parámetro *max_age* de la petición de autenticación, el cliente deberá comprobar el valor y en caso de que se haya superado el tiempo se tendrá que pedir una re-autenticación.

c) Tiene que validar el *Token* de Acceso:
Si el *ID Token* contiene el *claim iat_hash*, el cliente puede utilizarlo para validar el *Token* de Acceso.

- **Autenticación mediante Flujo Implícito (Implicit Flow)**
El flujo Implícito se utiliza principalmente por los clientes implementados en un navegador usando un lenguaje scripting. El Token de Acceso y el *ID Token* se devuelven directamente al cliente, que puede exponerlos al usuario final y a las aplicaciones que tienen acceso a su agente de usuario. Por ello, el servidor de autorización no implementa la autenticación del cliente.
Otra de las diferencias de este tipo de flujo frente al Flujo Código es que todos los *Tokens* son devueltos desde el Authentication Endpoint, por lo que el *Token Endpoint* no se utiliza. En cambio el *Authorization Endpoint* y el *Userinfo Endpoint* se siguen utilizando igual.
Pasos del Flujo Implícito: La *Figura 7* muestra el proceso de autenticación usando flujo implícito.

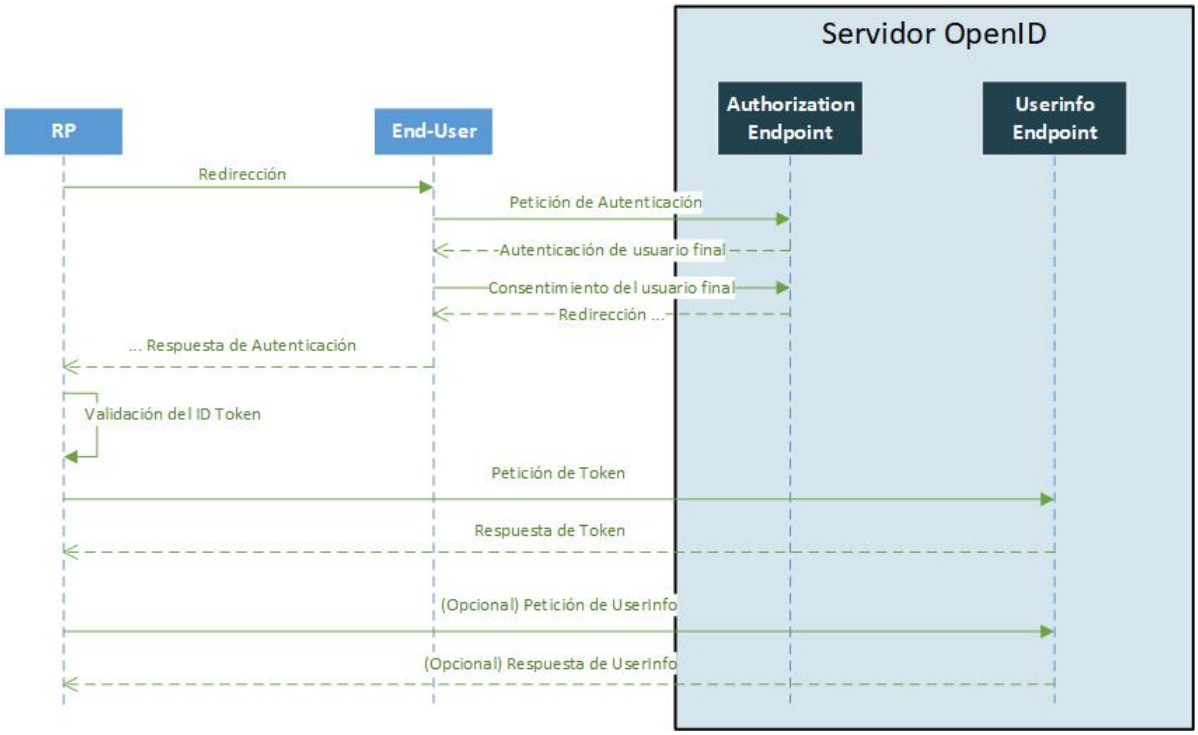

Figura 7: Flujo del Flujo Implícito

1. *Petición de autenticación (Authentication Request)*:
   Al igual que en el Flujo Código, es una petición que solicita que el usuario final esté autenticado por el proveedor de OpenID.
   Los *claims* utilizados en este flujo son los mismos mencionados en el Flujo Código, aunque los siguientes tienes distintas características:

Claim	requisito	Descripción
response_type	Obligatorio	Valor que determina el tipo de flujo de autenticación que se está utilizando. Cuando se utiliza el Flujo Implícito, entonces este valor tiene que ser *id_token token*
redirect_uri	Obligatorio	En este flujo, al tratarse de una aplicación nativa, entonces puede utilizar en el esquema de HTTP localhost como nombre de host, a diferencia de Flujo Código.
nonce	Obligatorio	Cadena de caracteres utilizada para asociar una sesión de Cliente con un *ID Token* y para mitigar los ataques de repetición.

Tabla 6: *Claims* de la petición de autenticación en Flujo Implícito

Un ejemplo de las peticiones de autenticación en el Flujo Implícito podría ser el siguiente, expuesto en las sus especificaciones OpenID Connect:

```
GET /authorize?
response_type=id_token %20token
&client_id=s6BhdRkqt3
&redirect_uri=https %3A %2F %2Fclient.example.org %2Fcb
&scope=openid %20profile
&state=af0ifjsldkj
&nonce=n-0S6_WzA2Mj HTTP/1.1
Host: server.example.com
```

La validación de las peticiones de autenticación es exactamente igual que en el Authentication Flow Code que se vio con anterioridad.

2. *El servidor de autorización autentica al usuario final*:
   La autenticación que realiza el servidor de autorización no varía en el resto de los flujos y se realiza de la misma manera que en la explicación del flujo explicado con anterioridad, apareciendo un menú donde se puede autenticar el usuario con las credenciales del servidor de autorización.
3. *El servidor de autorización obtiene el consentimiento del usuario final*:
   Igualmente, como pasaba con el anterior flujo, la autorización se realiza mediante la aparición de un cuadro interactivo que solicita la autorización por parte del usuario final.
4. *Respuesta de autenticación(Authentication Response)*:
   Una vez validada la petición de autenticación, la característica en este flujo de OpenID Connect es que el servidor de autenticación devuelve el *ID Token* y si se solicita también, el Token de Acceso. Los *claims* que se utilizan serán los mismos que en el Flujo Código pero con las siguientes variaciones:

Claim	requisito	Descripción
access_token	Opcional	Es el valor del Acces Token. Es devuelto siempre y cuando en la petición de autenticación se haya solicitado lo contrario, es decir, que el valor de *response_type* sea igual a id_token.
token_type	Obligatorio	Es el valor del Token Type, que tiene que ser igual a Bearer a menos que en la negociación con el servidor de autorización se haya solicitado otro valor.

Continuación de la tabla

Tabla 7 – Continuación de los *claims*

Claim	requisito	Descripción
id_token	Obligatorio	Es el valor del *ID Token*
state	Obligatorio si está incluido el parámetro state en la petición de autorización	El cliente debe de comprobar que el parámetro *state* en la petición de autenticación es igual a valor en la respuesta.
expires_in	Opcional	Tiempo de expiración del Token de Acceso, medido en segundos, desde que se generó la respuesta.

Tabla 7: *Claims* en la respuesta de autenticación, Flujo Implícito

Un ejemplo de respuesta en el Flujo Implícito puede ser el siguiente:

```
HTTP/1.1 302 Found
Location: https://client.example.org/cb#
access_token=SlAV32hkKG
&token_type=bearer
&id_token=eyJ0 ... NiJ9.eyJ1c ... I6IjIifX0.DeWt4Qu ... ZXso
&expires_in=3600
&state=af0ifjsldkj
```

Como se puede apreciar en el ejemplo, y a diferencia de lo que pasaba en el Flujo Código, no devuelve el parámetro *code*, que se utilizaba posteriormente en la petición de Token para conseguir el *ID Token* y el Token de Acceso. En este flujo se devuelve directamente el valor de id_token.
Dado que los parámetros de respuesta se devuelven directamente en el valor de la URI de redirección, es necesario que en el agente de usuario interprete y analice los valores codificados del fragmento y se lo pase al cliente para que pueda procesarlos y utilizarlos.
En el caso de haber un error en la autenticación, el mensaje de error sería similar a lo explicado en esta sección en el Flujo Código, a excepción de valores de los parámetros que tiene este flujo.
Para poder validar la respuesta de autenticación habría que realizar las siguientes comprobaciones:

a) Tiene que seguir las validaciones que se describen en la RFC6749 de Oauth 2.0.

b) Tiene que validar el *ID Token*:

En este tipo de flujo también hay una variación en las características de las *claims*, al igual que pasaba en el flujo anterior. Las diferencias en los valores respecto a lo comentado en el apartado de la explicación del *ID Token* son las siguientes:

Claim	requisito	Descripción
nonce	Obligatorio	Cadena de caracteres que sirve para asociar una sesión de cliente con un *ID Token*. En el Flujo Implícito su uso es obligatorio.
at_hash	Opcional	Su valor es la codificación de base64url de la mitad izquierda del hash de los octetos en ASCII del valor *access_token*, donde el algoritmo de hash utilizado es el algoritmo hash utilizado en el parámetro *alg* expuesto en el *ID Token*.

Tabla 8: *ID Token claims* en Flujo Implícito

La validación del *ID Token* es igual que en el caso del anterior flujo, a excepción de que habría que validad algunos casos particulares de la siguiente manera:

1) El Cliente debe validar la firma del *ID Token* de acuerdo con JWS, utilizando el algoritmo especificado en el parámetro *alg* de la cabecera JOSE. A diferencia del Flujo Código, no se puede usar la validación del servidor TLS para validar al emisor en vez de comprobar la firma.
2) El valor de la reclamación *nonce* se debe comprobar para verificar que es el mismo valor que el que se envió en la petición de autenticación.

c) Tiene que validar el Token de Acceso en el caso de que se haya pedido.
La validación del Token de Acceso es similar al anterior flujo a excepción de que el cliente tendrá que realizar siguientes pasos:

1) Se debe de utilizar una función resumen (hash) con los octetos de la representación ASCII de la *access_token* con el algoritmo de hash especificado en el parámetro *alg* de la cabecera JOSE del *ID Token*.

2) Tome la mitad izquierda del hash y codificarlo en base64url.
3) El valor de *iat_hash* en el *ID Token* tiene que coincidir con el valor conseguido en los anteriores pasos.

- **Autenticación mediante Flujo Híbrido (Hybrid Flow)**
  En el Flujo Híbrido todos los componentes del proveedor de OpenID entran en juego, y algunos *Tokens* son devueltos por el *Authorization Endpoint* y otros por el contrario por el *Token* Endpoint.
  Se podría decir que el Flujo Híbrido, tal como indica su nombre, en la autenticación puede comportarse como si fuera un Flujo Código o como un Flujo Implícito, dependiendo del valor o valores del parámetro *respons_type*.

1. *Petición de autenticación (Authentication Request)*:
   Este tipo de petición se envía para poder autenticar al usuario final, como pasaba con los anteriores flujos explicados, pero con la siguiente variante:

Claim	requisito	Descripción
response_type	Obligatorio	Valor del Tipo de respuesta que determina el tipo de flujo de autorización que se va a utilizar, incluidos los parámetros que se devuelven desde los puntos finales utilizados. Cuando se utiliza el Flujo Híbrido, este valor es *code id_token*,*code token* o *code id_token token*.

Tabla 9: *Claims* de la petición de autenticación en el Flujo Híbrido

Un ejemplo de una petición de autenticación en el Flujo Híbrido podría ser el siguiente, descrito en las especificaciones de OpenID Connect:

```
GET /authorize?
response_type=code%20id_token
&client_id=s6BhdRkqt3
&redirect_uri=https%3A%2F%2Fclient.example.org%2Fcb
&scope=openid%20profile%20email
&nonce=n-0S6_WzA2Mj
&state=af0ifjsldkj HTTP/1.1
Host: server.example.com
```

Pero ¿Cómo podemos saber cómo entregará el servidor el *ID Token*

y el Token de Acceso en el Flujo Híbrido?
Oauth 2.0 ha definido las respuestas[5] a los múltiples valores enviados en una petición de autenticación de la siguiente manera:

Valores de response_type	Descripción
code token	Cuando se envían estos valores en *response_type*, una respuesta válida debe de incluir un Token de Acceso, un *token_type* y un Código de Autorización. La respuesta por defecto de este tipo de petición es la codificación de fragmentos y la codificación de consultas no se debe de utilizar.
code id_token	Cuando se envían estos valores en *response_type*, una respuesta válida debe de incluir *id_token* y un Código de Autorización. La respuesta por defecto de este tipo de petición es la codificación de fragmentos y la codificación de consultas no se debe de utilizar.
id_token token	Cuando se envían estos valores en *response_type*, una respuesta válida debe de incluir un Token de Acceso, un *token_type* y un id_token. La respuesta por defecto de este tipo de petición es la codificación de frangmentos y la codificación de consultas no se debe de utilizar.
code id_token token	Cuando se envían estos valores en *response_type*, una respuesta válida debe de incluir un Código de Autorización, un *id_token*, Token de Acceso y un *token_type*. La respuesta por defecto de este tipo de petición es la codificación de frangmentos y la codificación de consultas no se debe de utilizar.

Tabla 10: Valores de response_type en el Flujo Híbrido

Dependiendo de los valores de *response_type* el *ID Token* y el Token de Acceso lo entregará el *Authorization Endpoint* o el *Token Endpoint*. Por esta razón este flujo se le llama híbrido, ya que puede actuar de forma similar a los otros flujos dependiendo de los valores que devuelva el servidor en el parámetro *response_type*.

Esto sería todo el proceso de autenticación que seguiría el cliente y el servidor según cada uno de los flujos.
Respecto a OpenID Connect nos falta aún hablar de otro tipo de petición igual de importante, pero totalmente opcional.

- **Petición UserInfo:**
  El *Userinfo Endpoint* es un recurso protegido OAuth 2.0 que devuelve *claims* sobre el usuario final autenticado. Para obtener las *claims* solicitadas sobre el usuario final, el cliente hace una solicitud utilizando un Token de Acceso obtenido mediante la autenticación de OpenID Connect y este se lo devuelve en formato JSON.

  1. *Petición de UserInfo*
     El cliente envía la Petición UserInfo utilizando los métodos de HTTP, GET o POST, pero se recomienda que la solicitud utilice el método HTTP GET y que el Token de Acceso se envíe utilizando el campo de cabecera de Autorización. Este sería un ejemplo de este tipo de petición expuesto en las especificaciones de OpenID Connect:

     ```
     GET /userinfo HTTP/1.1
     Host: server.example.com
     Authorization: Bearer SlAV32hkKG
     ```

     Como se puede observar la petición se envía con el mismo formato que un Token Bearer, especificado en *OAuth 2.0 Bearer Token Usage [RFC6750]* y han utilizado el método GET.
  2. *Respuesta de UserInfo*
     Una vez se haya podido comprobar que la petición es válida, cuyo único requisito obligatorio es el comprobar que la comunicación está cifrada mediante TLS, se puede recibir este tipo de respuesta afirmativa:

     ```
     HTTP/1.1 200 OK
     Content-Type: application/json


     "sub": "248289761001",
     "name": "Jane Doe",
     "given_name": "Jane",
     "family_name": "Doe",
     "preferred_username": "j.doe",
     "email": "janedoe@example.com",

     "picture": "http://example.com/janedoe/me.jpg"
     ```

     Por otro lado si la petición ha sido incorrecta se recibirá por ejemplo la siguiente respuesta:

     ```
     HTTP/1.1 401 Unauthorized
     WWW-Authenticate: error"invalid_token",
     error_description="The Token de Acceso expired"
     ```

Con esto finalizamos la explicación sobre OpenID Connect, un protocolo completo y sencillo gracias al uso de JSON y de tener unos flujos de petición intuitivos y seguros.

2.1.2. SAML

SAML, acrónimo de Security Assertion Markup Language, es un protocolo de autenticación de usuario basado en XML.
Es un estándar que se inició en el año 2001 y que su primera versión fue liberada en el 2005, por lo que ya lleva bastantes años siendo utilizando por grandes empresas y ha sido estandarizado como protocolo de gestión de identidad. También tiene integradas una multitud de librerías que le otorgan integridad y robustez.

- **Principales actores de SAML**
  Se pueden observar tres grupos diferenciados a la hora de que un usuario realice la autenticación, y son los siguientes:

  - **Cliente**:
    Usuario final que intenta acceder a la aplicación utilizando unas credenciales que mantiene en otra aplicación, siempre a través de un agente de usuario como el navegador, dispositivo móvil, escritorio virtual, etc.
    En la infraestructura de gestión de identidad descrita en el apartado **2.1. Infraestructura de gestión de identidad** este rol equivaldría al de usuario final.
  - **Proveedor de servicios (SP):**
    Es el rol equivalente al mismo con este nombre en la infraestructura de gestión de identidad, y su función es la de dar acceso al usuario o entidad como se ha descrito en el apartado **2.1. Infraestructura de gestión de identidad**.
  - **Proveedor de identidad (IdP):**
    Este rol sería el equivalente al rol de IdP genérico de una estructura de gestión de identidad. Llevaría a cabo las funciones de gestión del usuario, tales como almacenamiento de credenciales, configuración de la cuenta de usuario, etc.

  A diferencia de los flujos de OpenID Connect que siguen siempre la misma estructura y el mismo flujo, en el caso de SAML cualquiera de los proveedores, o el de identidad o el de servicio, puede iniciar la conexión dependiendo de lo que se esté solicitando.
  Flujo general de SAML
  Ahora que se conoce mejor el protocolo SAML y qué componentes lo caracterizan se va a explicar cómo realizar de forma segura el intercambio de asertos entre el IdP y SP utilizando un *binding* definido:

Figura 8: Diagrama de flujo general de SAML

Se puede observar en la **Figura 9** que el flujo mantenido entre los siguientes roles sigue el siguiente patrón:

1. El usuario de agente envía una petición vía HTTP al *SP* solicitando un recurso de destino.
   El proveedor de servicios realiza una comprobación de seguridad en nombre del recurso de destino, pero si ya existe un contexto de seguridad válido en el proveedor de servicios, entonces se omiten los pasos 2 y 7.
2. El *SP* redirecciona al agente de usuario hacia el servicio de autenticación en el *IdP* mediante *XHTML*.
3. El agente de usuario realiza una petición al servicio de autenticación del *IdP* y se autentica en el proveedor.
4. Una vez que el servicio de autenticación valida al usuario, envía una respuesta que contiene un documento en formato XHTML al agente de usuario.
5. El agente de usuario envía una petición POST de HTTP al *SP* para conocer los valores de los asertos.
6. El *SP* procesa la respuesta, creando un contexto de seguridad y redirige al agente de usuario al recurso de destino
7. El agente de usuario hace una petición al recurso para consultarlo que se encuentra en el *SP*.
8. El *SP* finalmente envía la información del recurso al agente de usuario.

Principales definiciones de SAML

Una vez conocidos los roles que tiene el protocolo y como afectan a la hora de la autenticación, debemos describir los principales componentes del recurso que le caracterizan antes de entrar en los detalles de los flujos de intercambio de información entre los distintos roles[8]:

- **Aserto (assertions):** contienen información de atributo, autenticación y autorización, que son los tres tipos diferenciados de aserto, de un cliente en particular y nos sirven para definir formas de afirmaciones de seguridad a la hora de realizar la autenticación. Están definidas mediante un esquema XML y pueden ser enviadas a raíz de una petición:
  1. **Aserto de atributo:** son afirmaciones que nos indican algún atributo del cliente, como puede ser por ejemplo su nombre, email, etc.
  2. **Aserto de autenticación:** son afirmaciones enviadas por el proveedor que realiza la autenticación al usuario y contienen información relevante a cerca de la autenticación, como puede ser la duración de validez, el sujeto autenticado, etc.
  3. **Aserto de autorización:** son afirmaciones que recogen información sobre lo que se le permite realizar al usuario, como el acceso a recursos por ejemplo.

  Un ejemplo extraído del documento de las especificaciones funcionales[9] de cómo sería la estructura de un aserto y para conocer cómo se integra en un esquema XML, sería el que se muestra en la **Figura 8**:

```
<Assertion ID="_a75adf55-01d7-40cc-929f-dbd8372ebdfc"
   IssueInstant="2003-04-17T00:46:02Z" Version="2.0"
   xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
   <Issuer>
      example.com
   </Issuer>
   <Subject>
      <NameID
        Format=
        "urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress">
        Alice@example.com
      </NameID>
      <SubjectConfirmation
        Method="urn:oasis:names:tc:SAML:2.0:cm:sender-vouches"/>
   </Subject>
   <Conditions NotBefore="2003-04-17T00:46:02Z"
               NotOnOrAfter="2003-04-17T00:51:02Z">
      <AudienceRestriction>
         <Audience>
            example2.com
         </Audience>
      </AudienceRestriction>
   </Conditions>
   <AttributeStatement>
      <saml:Attribute
   xmlns:x500=
     "urn:oasis:names:tc:SAML:2.0:profiles:attribute:X500"
   NameFormat=
     "urn:oasis:names:tc:SAML:2.0:attrname-format:uri"
   Name="urn:oid:2.5.4.20"
   FriendlyName="telephoneNumber">
         <saml:AttributeValue xsi:type="xs:string">
               +1-888-555-1212
         </saml:AttributeValue>
      </saml:Attribute>
   </AttributeStatement>
</Assertion>
```

Figura 9: Ejemplo de aserto en SAML

- **Protocolos (protocols):** son unos protocolos definidos por SAML que especifican cómo se solicitan o responde a los asertos mediante un mecanismo de intercambio de información utilizando estructuras XML. SAML define los siguientes protocolos para su estándar:
  - **Assertion Query and Request Protocol:** este protocolo define los mensajes y las reglas de procesamiento para solicitar los asertos existentes.
  - **Authentication Request Protocol:** cuando un servidor desea obtener asertos que contengan declaraciones de autenticación para establecer un contexto de seguridad puede utilizar este protocolo para realizar las peticiones y recibir a cambio una o más asertos.
  - **Artifact Resolution Protocol:** este protocolo suministra un me-

canismo que nos permite enviar los mensajes de SAML utilizando un enlace SAML por referencia en lugar de por valor.

- **Name Identifier Management Protocol:** es un protocolo que nos permite modificar el nombre del identificador cuando el *IdP* desea cambiar el valor y/o el formato del identificador.
- **Single Logout Protocol:** es un protocolo mediante el cual permite a la autoridad pertinente cerrar las sesiones particulares de forma casi simultánea.
- **Name Identifier Mapping Protocol:** es un protocolo que permite el cambio de formato de un nombre de identificador cuando se solicita.

- **Bindings:** es el método utilizado por SAML para parsear el contenido de las peticiones HTTP o SOAP a esquemas XML para el intercambio de asertos. Existen actualmente seis tipos distintos de *bindings*, que son los siguientes:
  1. **SAML SOAP Binding:** especifica cómo conseguir mensajes SAML contenidos en peticiones SOAP.
  2. **HTTP Redirect Binding:** especifica como utilizar peticiones de redirección HTTP para transportar mensajes SAML codificados en base64.
  3. **HTTP POST Binding:** especifica cómo utilizar peticiones POST para el envío de información SAML dentro del contenido de un formulario HTML.
  4. **HTTP Artifact Binding:** especifica el envío en una petición HTTP de una referencia a una petición SAML.
  5. **Reverse SOAP (PAOS) Binding:** permite que un solicitante de una petición HTTP actúe como procesador de mensajes SOAP que contengan mensajes SAML.
  6. **SAML URI Binding:** especifica cómo las peticiones SAML transportan contenidos en URIs.
- **Profiles:** los protocolos, *bindings* y asertos de SAML no se pueden utilizar por sí solos, se necesitan combinar para formar perfiles. Se definen como la secuencia de asertos, *bindings* y protocolos para realizar y conseguir cada uno de los casos de uso de SAML. Los perfiles definidos por el estándar son los siguientes:
  - **SSO Profiles:** especifica cómo se puede proporcionar un servicio de firma simple mediante el uso de un navegador web y utilizando peticiones de protocolo combinadas con los *bindings HTTP Redirect*, *HTTP POST* y *HTTP Artifact*. Está configurado por un subconjunto de perfiles que tratan entre sí para el correcto funcionamiento del mismo.
  - **Artifact Resolution Profile:** protocolo de resolución de artefactos que utiliza combinado con el *HTTP Artifact Binding*.
  - **Assertion Query/Request Profile:** define cómo debe utilizar el protocolo SAML el *SAML SOAP Binding*.

- **Name Identifier Mapping Profile:** este perfil está diseñado para soportar las conexiones a través de un dispositivo móvil ya que especifica las peticiones del protocolo y utiliza el *Reverse SOAP (PAOS) Binding*.
- **SAML Attribute Profiles:** define los mecanismos que utiliza un *SP* para conocer que *IdP* está utilizando el usuario.

Con los anteriores apartados se han abarcado las características más importantes del estándar de SAML, quedando demostrado que es un protocolo robusto y seguro.

2.2. Servicios web

En este apartado se describen los distintos servicios web más importantes que tenemos que conocer para entender mejor el plugin y el funcionamiento de los protocolos de autenticación.
Algunos de los apartados ya hicieron referencia a alguno de los servicios web y se han descrito de forma resumida, pero aquí entraremos en detalle haciendo hincapié en las propiedades de cada uno de los que son más relevantes en este proyecto.

2.2.1. REST

REST, acrónicmo de Representational State Transfer, es un conjunto de arquitecturas Software que siguen un estilo de diseño y que permiten diseñar páginas web, siguiendo un conjunto de claves fundamentales que orientan en su metodología para compartir recursos y envío de estos a otros clientes[7].
En cierto modo se utiliza REST para definir de forma general cualquier principio de arquitectura, pero hoy en día se usa directamente para definir sistemas que utilizan el protocolo HTTP, que es el protocolo más utilizado de este tipo de arquitectura. Para obtener esta robustez en REST se han seguido unos principios que habría que seguir para definir un conjunto de arquitecturas web como RESTful.

- **Claves fundamendales de REST:**
  - **Sistema Cliente-Servidor:**
    Para cumplir con los principios de la arquitectura REST se debe distinguir dos roles principales, el rol de cliente y de servidor. Ambos tienen que ser independientes uno del otro:
    - Servidor: la responsabilidad de este rol sería el de generar respuestas a las peticiones recibidas, incluyendo toda la información relativa a la petición además de incluir enlaces para que se pueda navegar de una forma fluida. También tiene que entender e interpretar las cabeceras del mensaje para poder ofrecer la solución óptima, como es el caso de las cabeceras caché que ayudan a reducir el tiempo de respuesta reduciendo las peticiones duplicadas.
    - Cliente: la responsabilidad de este rol sería la de generar peticiones de consulta al servidor con toda la información incluida, dirección y recursos a los cuales intenta tener acceso, y la de comprobar las

cabeceras importantes de las peticiones que le llegan. Tal como se comentó en el rol anterior, uno de los principales sería la cabecera caché, ya que en este caso tendría que realizar una copia local de la información para no generar peticiones repetidas.

Ambos roles trabajan de una forma fluida para proporcionar al usuario final información robusta y sin errores ni fallos de seguridad.
Un ejemplo de diagrama de arquitectura REST sería el siguiente:

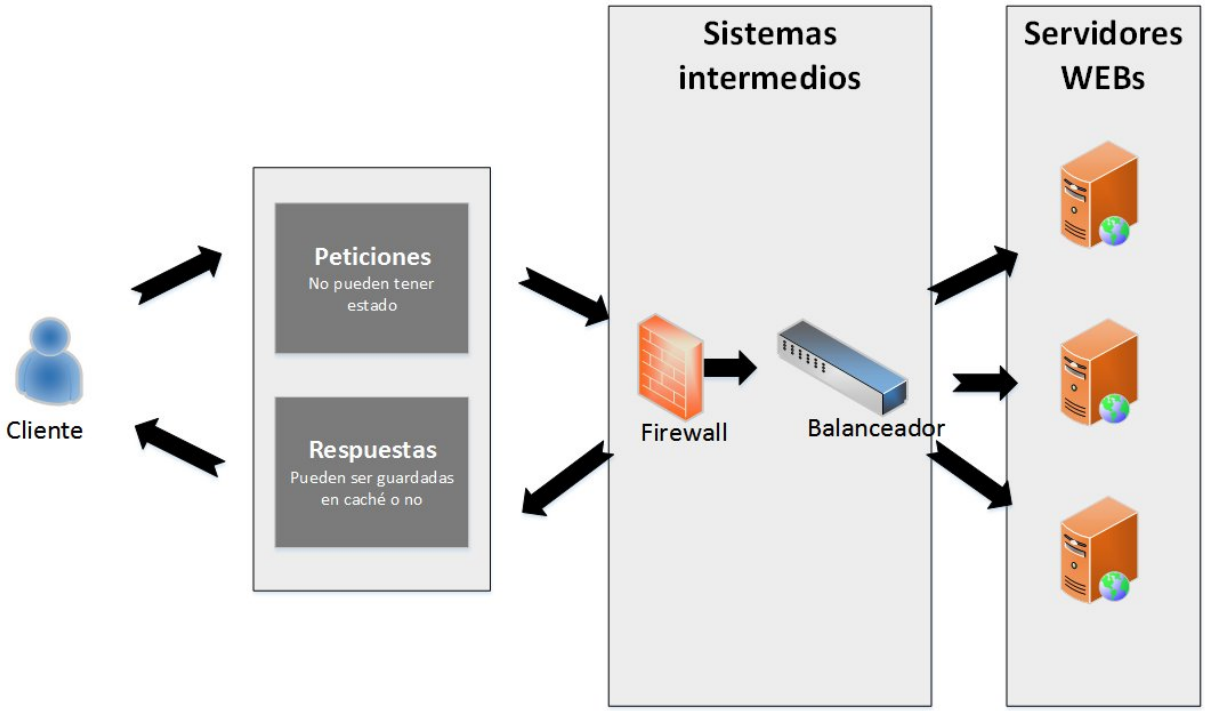

Figura 10: Ejemplo de arquitectura REST

Más adelante explicaremos la estructura de las respuesta y las peticiones que se realizan en el protocolo HTTP, para que se tenga un concepto más claro de la información que comparten.

- **Protocolo sin estado:**
  Debido al gran avance que existe hoy en día en los servicios web, muchas veces requieren, aparte de servidores finales que contengan la información y a los que habría que enviar las peticiones, clústers, servidores intermedios y proxies para reducir el tiempo de respuesta entre las peticiones. Por esta razón, es necesario que las peticiones que se envían sean comprensibles y que contengan toda la información necesaria para que los equipos intermedios puedan gestionar las peticiones sin necesidad de guardar datos localmente ni los estados de las peticiones.
  La teoría sería la anterior, pero es cierto que hoy en día, en la práctica se utilizan cookies para guardar los estados y contenidos de algunas páginas.

- **Métodos bien definidos:**
  El tercer principio clave para el funcionamiento de REST, es que tiene que tener definidos métodos de forma clara, para que se puedan aplicar a todos las peticiones, y donde el cliente y el servidor puedan comprender su funcionamiento.
  Por ello HTTP tiene implementados varios métodos que realizan peticiones uno a uno, es decir, por cada una de las llamadas a los métodos se realiza una acción recíproca, brindando estabilidad al protocolo y comprensión, ya que es necesario realizar la petición utilizando los métodos de forma explícita.
  Las principales funciones definidas en HTTP son las siguientes:
  - PUT: función responsable de actualizar, subir o cambiar un recurso específico en un servidor.
  - POST: función responsable de enviar la información en el cuerpo del mensaje, para que sea procesada por uno de los roles anteriormente definidos.
  - GET: función responsable de representar el recurso específico al que se quiere consultar. Es un método poco seguro al incluir toda la petición de la consulta en el campo URI, visible a cualquiera que puede interceptarlo.
  - DELETE: función responsable de la eliminación de un recurso concreto.
- **Utilización de directorios en forma de URI:**
  Para poder crear esa fluidez entre las peticiones, todas ellas tienen que incluir un enlace URI para poder redireccionar y conseguir recursos de los servidores. Las URIs siguen una estructura jerárquica lo que facilita su lectura y otorga rapidez al protocolo. Los directorios URI pueden entenderse mejor si pensamos en ellos como ramas que van desglosándose a partir de una raíz.
  Por ejemplo la dirección *www.ejemplo_uri.com/raiz/hijo1* podemos observar que la raíz sería el directorio */raíz* y una de sus posibles ramas sería */hijo1*, que tal vez genere en más ramas secundarias.
- **Utilización de hipervínculos:**
  La última clave de un sistema REST es que utiliza hipervínculos para realizar otras acciones sobre determinados datos, que podrían ser transmitidos en formato XML, JSON o HTML. Utilizando esta forma de transmisión de datos hace más legible la información y más fácil de tratarla en las distintas peticiones.
  Más adelante, en el apartado **2.2.2. JSON** se describe uno de los principales formatos que va ganando adeptos día tras día.

Siguiendo estas claves, se puede considerar que el sistema cumple con los requisitos de ser un conjunto de arquitecturas REST.

- **Estructura de las peticiones REST:**
  En los puntos tratados, comentamos la necesidad de que siguiera una

estructura sin estado y que tuviera toda la información contenida en cada una de las peticiones, por ello en este apartado se va a explicar cómo son las peticiones realizadas y su respuesta con un ejemplo simple:

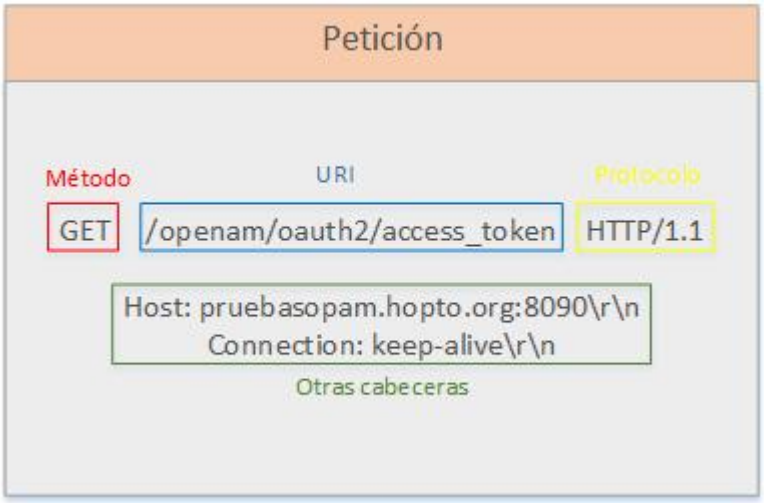

Figura 11: Ejemplo de petición REST

La **Figura 11** muestra es un ejemplo de petición REST y se puede observar que una misma petición contiene todos los datos relevantes de la misma:

- Método: método con el que se va a realizar la petición, en este caso *GET*
- URI: URI a la que se le realiza la petición.
- Protocolo: protocolo con el que se hace la petición, *HTTP/1.1*
- Cabeceras: cabeceras importantes para la petición, como son la de *HOST*, obligatoria en *HTTP/1.1* y la de *Connection* que indica que la conexión la desea persistente.

Una posible respuesta a esta petición podría ser la mostrada en la **Figura 12**:

Figura 12: Ejemplo de respuesta REST

Se pueden observar en la respuesta los siguientes campos importantes:

- Protocolo: al igual que en el caso de petición se indica el protocolo de la respuesta.
- Estado: indica el estado de la petición, que en este caso ha sido un *OK*.
- Cabeceras: cabeceras importantes para la petición, en este caso las cabeceras *Cache-Control* y *Pragma* para que no guarden el contenido confidencial de la respuesta al contener un Token de seguridad, *Server* para indicar el tipo de servidor empleado y *Content-Type* que indica el tipo de contenido que va adjunto a la respuesta.
- Contenido: aquí se adjunta el contenido en caso de haberlo, y en nuestro ejemplo es de tipo JSON.

Con estos puntos se ha explicado la importancia y el comportamiento de un sistema REST en una arquitectura global.

2.2.2. JSON

JSON, acrónimo de JavaScript Object Notation (Notación de objetos JavaScript) es un formato de texto ligero que ayuda a intercambiar información ya que utiliza una sintaxis muy intuitiva y fácil de usar[10].
Al principio nació como una alternativa a XML, pero gracias a su éxito influenciado principalmente por JavaScript ya se considera un lenguaje de programación independiente.

Ventajas de JSON frente a otros formatos:

- La mayor ventaja que tiene JSON frente a otros formatos de intercambio de datos, es que puede ser leído por distintos lenguajes de programación (*C, PHP, RUBY, Java, PERL, etc*) ya que es independiente del lenguaje.
  Este hecho ha permitido que tenga cada vez más seguidores y a la vez

ayuda al intercambio de información entre aplicaciones muy diferenciadas, convirtiéndolo en un lenguaje de programación flexible.

- Trabaja con un gran abanico de tipos de datos:
  - Número: representa un número que puede ser entero (integer) o fraccionado (float)
  - Cadenas de caracteres: cadenas de uno o más caracteres que está entrecomillado.
  - Booleanos: valor que puede ser o *true* o *false*.
  - conjunto: conjunto de variables ordenadas y separadas por comas. Las variables pueden ser tanto tipo cadena de caracteres, como números o booleanos, y vienen siempre entre corchetes.
  - Objetos: es un conjunto de pares definido de la la forma *nombre:valor*, que al contrario que el conjunto está desordenado y siempre viene entre llaves. Cada par de nombre y valor viene separado de otro con comas.
  - null: valor nulo.
- Al tratarse de un lenguaje que utiliza un par de clave y valor, a nivel de usuario resulta más fácil e intuitivo el poder escribirlo y leerlo.

Un ejemplo de un objeto JSON sería el que se muestra en la **Figura 13**:

```
{"Grado Telecomunicaciones":
 [
        {"Telemática":
        [
                {"Asignatura":"Programación","Alumnos":35},
                {"Asignatura":"Cálculo","Alumnos":60},
        ]
        },
        {"Sistemas de Comunicaciones":
        [
                {"Asignatura":"Álgebra","Alumnos":65},
                {"Asignatura":"Antenas","Alumnos":15},
        ]
        }
 ]
}
```

Figura 13: Ejemplo de JSON

Como se puede observar se ha creado un objeto llamado "Grado Telecomunicaciones" y dentro se ha añadido un array con dos elementos, "Telemática" y "Sistemas de Comunicaciones".
El elemento llamado "Telemática" almacena un array con con otros dos elementos

Asignatura, que contienen el nombre y la cantidad de alumnos que tiene cada una de ellas:

- {"Asignatura":"Programación","Alumnos":35}
- {"Asignatura":"Cálculo","Alumnos":60}

Cada uno de los elementos anteriores está formados por dos pares de clave y valor. Con este formato resulta fácil entenderlo en lenguaje humano, lo que facilita su uso y que se haya extendido de una forma tan rápida entre los usuarios.

2.2.3. JWT

JWT, acrónimo de JSON Web Tokens, es un tipo de autenticación basado en *Tokens*, lo cual es una forma bastante compacta y segura de transferir los *claims* de servidor a cliente.
Los *claims* en un JWT se codifican como un objeto JSON permitiendo que los *claims* sean firmados digitalmente o protegidos con un código de autenticación de mensajes (MAC) o cifrado directamente el contenido.
El formato de un JWT está compuesto por tres segmentos codificados en base64url separados por los caracteres de punto ('.') [11]:

1. Primer segmento:
   El primer segmento representa la cabecera JOSE, compuesta por parámetros identificativos. Los relevantes en nuestro documento son los siguientes:
   - typ: indicador que sirve para declarar que ese tipo de dato es un JWT.
   - kid: es un identificador clave utilizado en la identificación de la clave para verificar la firma.
   - alg: representa el algoritmo con el que ha sido cifrado el JWT.
2. Segundo segmento:
   El segundo segmento representa la carga útil (payload), es decir, la parte donde están codificados los *claims* en JWT. Ya se ha hablado en el apartado **2.1.1. JWT**. OpenID Connect de los principales *claims* que se van a utilizar en nuestro protocolo.
3. Tercer segmento:
   El tercer segmento representa la firma. Está formada por la parte de la cabecera JOSE y el payload, cifrada en Base64 con una clave secreta.

Una vez conocidas todas las componentes de JWT, se puede observar en este ejemplo lo explicado para que quede más claro:

- Ejemplo de un JWT codificada:

```
eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ.ewogI
mlzc yI6ICJodHRwOi8vc2VydmVyLmV4YW1wbGUuY29tIiwKICJ
zdWIiOiAiMjQ4Mjg5NzYxMDAxIiwKICJhdWQiOiAiczZCaGRSa3
F0MyIsCiAibm9uY2UiOiAibi0wUzZfV3pBMk1qIiwKICJleHAiOi
AxMzExMjgxOTcwLAogImlhdCI6IDEzMTEyODA5NzAKfQ.ggW8
hZ1 EuVLuxNuuIJKX_V8a_OMXzR0EHR9R6jgdqrOOF4daGU96Sr
_P6qJp6IcmD3HP99Obi1PRs-cwh3LO-
p146waJ8IhehcwL7F09JdijmBqkvPeB2T9CJNqeGpe-
gccMg4vfKjkM8FcGvnzZUN4_KSP0aAp1tOJ1zZwgjxqGByKHiO
tX7TpdQyHE5lcMiKPXfEIQILVq0pc_E2DzL7emopWoaoZTF
_m0_N0YzFC6g6EJbOEoRoSK5hoDalrcvRYLSrQAZZKfly
uVCyixEoV9GfNQC3_osjzw2PAithfubEEBLuVVk4XUVrWO
LrLl0nx7RkKU8NXNHq-rvKMzqg
```

- Primera parte codificada:

```
eyJhbGciOiJSUzI1NiIsImtpZCI6IjFlOWdkazcifQ
```

Su valor descodificado sería: {"alg":"RS256","kid":"1e9gdk7"}
Indica, aparte del *kid*, el algoritmo utilizado para securizar el JWT, que en este caso y según las especificaciones de JWT, sería un RSA usando SHA-256 como algoritmo Hash.

- Segunda parte codificada:

```
ewogImlzcyI6ICJodHRwOi8vc2VydmVyLmV4YW1wbGUuY29tIi
wKICJzdWIiOiAiMjQ4Mjg5NzYxMDAxIiwKICJhdWQiOiAiczZC
aGRSa3F0MyIsCiAibm9uY2UiOiAibi0wUzZfV3pBMk1qIiwKI
CJleHAiOiAxMzExMjgxOTcwLAogImlhdCI6IDEzMTEyODA5Nz
AKfQ
```

Su valor codificado sería:

```
{
"iss": "http://server.example.com",
"sub": "248289761001",
"aud": "s6BhdRkqt3",
"nonce": "n-0S6_WzA2Mj",
"exp": 1311281970,
"iat": 1311280970
}
```

Estos serían todos los *claims* utilizados para la autenticación mediante *Tokens* de JWT.

- Tercera parte codificada:

```
ewogImlzcyI6ICJodHRwOi8vc2VydmVyLmV4YW1wbGUuY29tIi
wKICJzdWIiOiAiMjQ4Mjg5NzYxMDAxIiwKICJhdWQiOiAiczZC
aGRSa3F0MyIsCiAibm9uY2UiOiAibi0wUzZfV3pBMk1qIiwKI
CJleHAiOiAxMzExMjgxOTcwLAogImlhdCI6IDEzMTEyODA5Nz
AKfQ
```

Esta sería la firma cifrada, que al tratarse de un algoritmo RSA necesitaríamos una clave pública o un certificado y una clave privada para poder

descifrarla. En cambio, si se utiliza un protocolo más sencillo como es el caso del HS256, únicamente hubiéramos necesitado un método de descifrado junto a la clave secreta para conseguir el contenido de la firma.

2.3. Tecnologías

En este apartado se hablará de las distintas tecnologías utilizadas en el desarrollo del plugin y la infraestructura necesaria para poder realizar las peticiones descritas en el apartado de OpenID Connect, además de las que se han utilizado para realizar el plugin y el seguimiento de la trazabilidad.

2.3.1. OpenAM

Figura 14: Servidor OpenAM

OpenAM, acrónimo de Open Access Manager, es un programa de gestión de accesos de código abierto, en el cual viene implementado el proveedor de OpenID Connect y reconoce los distintos flujos del protocolo.
Se ha escogido este proveedor frente a otros por tratarse de un servidor completo donde se le pueden realizar todo tipo de configuraciones, como puede ser la configuración del mismo servidor, de los agentes, los usuarios y el nivel de securización de las comunicaciones; además de que disponía de una versión gratuita para realizar las pruebas, disponer del perfecto entorno para nuestro plugin y por estar certificado como implementación de OpenID Connect en la web oficial del protocolo[12].

OpenAM será utilizado como Servidor OpenID Connect en el desarrollo de la infraestructura de pruebas, ya que dispone de un *Token Endpoint*, *Authorization Endpoint* y *Userinfo Endpoint*, todo en uno, y permite configurar los usuarios

en la forma que lo deseemos, añadiendo algunos nuevos desde cero, creando sus respectivos *claims* para que sean intercambiados con el cliente. Para la realización

de las pruebas y para crear un entorno de cliente-servidor, OpenAM facilita también un cliente gratuito:

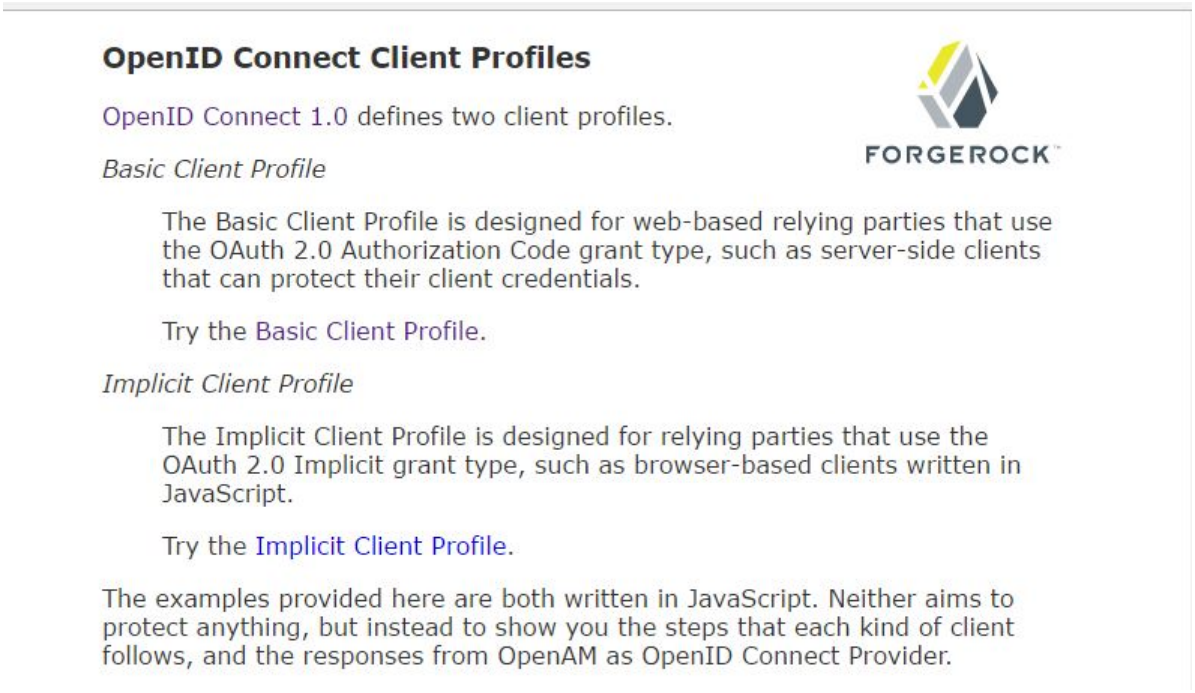

Figura 15: Cliente facilitado por OpenAM

Como se puede observar en la **Figura 15**, permite generar los flujos de el Flujo Código y el Flujo Implícito para autenticarnos en nuestro Servidor OpenAM, lo que facilita desarrollar el plugin usando un único cliente. También da la opción de poder autenticarnos al Servidor OpenAM mediante un dispositivo móvil, lo que otorga un mejor escenario de pruebas para el desarrollo de un plugin eficiente y que contemple las máximas posibilidades posibles.
Por todas estas razones, se ha escogido OpenAM para implementar la infraestructura cliente - servidor, y más adelante, en la sección **4. Arquitectura de despliegue**, se habla de la instalación de la infraestructura y de las trazas intercambiadas entre ambas partes.

2.3.2. Wireshark

Wireshark es un analizador de protocolos de red, utilizado como herramienta de análisis y de depuración de protocolos, para desarrollo tanto de éstos como de software relacionado, y que sirve también como herramienta de aprendizaje. Es una herramienta de software libre que progresa gracias a los donativos de expertos en redes de todo el mundo y es la continuación de un proyecto iniciado por Gerald Combs en 1998, con su título original *Ethereal*.
Esta herramienta es similar a otras existentes en el mercado, como es el caso de

tcpdump, pero gracias a su interfaz gráfica es más fácil de utilizar y es más intuitiva como se puede observar en la **Figura 16**:

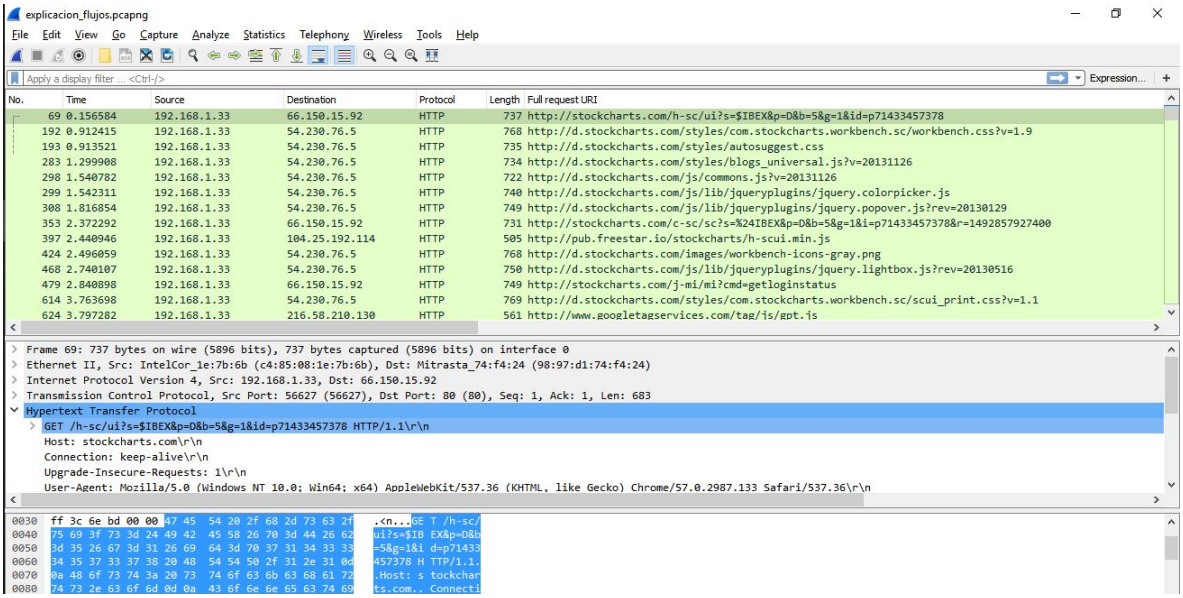

Figura 16: Dashboard de Wireshark

Wireshark ofrece mutitud de ventajas descritas en su página web oficial[13], entre las que destacan las siguientes:

1. Puede inspeccionar profundamente más de 480 protocolos de red, y cada día más ya que se pueden desarrollar de forma abierta nuevos analizadores.

2. Permite analizar tráfico real y también tráfico guardado previamente en capturas en cualquier momento.

3. Tiene multitud de filtros que ayudan a afinar la búsqueda de las trazas que interesan.

4. Puede leer archivos de captura de distintos productos.

5. Para poder guardar el contenido se puede exportar a XML, PostScript, CSV, o incluso texto plano.

6. Su buscador está basado en tres paneles para realizar y entender mejor las búsquedas del tráfico filtrado, como se puede observar en la **Figura 16**.

7. Es multiplataforma, funciona para Windows, Linux, macOS, Solaris, FreeBSD, NetBSD y otros.

8. Reconstruye sesiones TCP cuyo mensaje ha sido fragmentado.

9. Además de realizar búsquedas en la interfaz gráfica, se puede realizar por línea de comendo gracias a una herramienta que viene incluida en Wireshark, Tshark.

10. Se pueden crear nuevos plugins "diseccionando" nuevos protocolos, o modificando uno ya creado y en distintos lenguajes de programación.

11. El tráfico se puede leer desde distintos tipos de red, ethernet, wifi, PPP y loopback.

Gracias a que Wireshark tiene esta multitud de ventajas se ha escogido para crear el plugin.
En el punto 10 se ha hablado de que Wireshark soportaba multitud de lenguajes para la creación de los plugins, o Disectores, por ello se va a realizar una introducción sobre LUA, el lenguaje de programación que se ha utilizado para el desarrollo de nuestro plugin.

- **LUA**
  Según la web oficial de LUA [14], *Lua es un lenguage de programación extensible diseñado para una programación procedimental general con utilidades para la descripción de datos. También ofrece un buen soporte para la programación orientada a objetos, programación funcional y programación orientada a datos. Se pretende que Lua sea usado como un lenguaje de script potente y ligero para cualquier programa que lo necesite. Lua está implementado como una biblioteca escrita en C limpio (esto es, en el subconjunto común de ANSI C y C++).*

  - **Conceptos básicos de LUA**
    La programación en LUA es un lenguaje muy sencillo que tiene conceptos muy parecidos a C, pero difiere en lo siguiente:
    - En LUA las variables no tienen tipo, pero los datos sí pueden ser nil, booleanos, números, cadenas de caracteres, funciones, userdata, hilos y tablas.
    - Es un lenguaje de programación que no necesita de compilación para poder ser ejecutado.
    - Las condiciones, bucles y funciones no necesitan llaves para saber por donde están delimitadas, únicamente necesitan terminar con un *end*.
    - El final de cada sentencia no necesita punto y coma para terminarlas.
    - No necesita que los programas tengan una función o método para que el programa funcione.
    - Todas las variables son globales, a menos que las predefinamos con *local*.

  - **LUA en Wireshark**
    Una vez conocidos los conceptos básicos de LUA, podemos empezar a explicar cómo interactúa LUA con Wireshark para poder analizar el tráfico recogido por Wireshark.
    Se puede programar de tres formas distintas en LUA para analizar tráfico en Wireshark, dependiendo de las características de nuestro plugin:

- Disector (Dissector)
  Un disector es utilizado para analizar paquetes de datos, por lo que es ideal su uso a la hora de analizar trazas desde cero sin que se base en otro analizador generado por Wireshark para realizar la disección de la traza.
- Post-Disector (Post-Disector)
  Un post-disector es un disector que se llama después de que cada distinto disector ya ha sido llamado. Son prácticos a la hora de programarlos ya que todos los campos necesarios para el protocolo ya existen para poder ser accedidos y poder así agregarlos al árbol de disección.
- Oyente (Listener) Un Listener se utiliza para recolectar datos después de que un paquete ha sido diseccionado. Es el más simple de los tres y únicamente se utiliza para mostrar información sin tratar previamente.

- **Ventajas y desventajas de utilizar LUA**
  A la hora del desarrollo del plugin tuvimos en cuenta los siguientes factores para escoger el lenguaje de programación LUA sobre C:
  - Ventajas:
    1. Es un protocolo sencillo de implementar y para realizar pruebas.
    2. No necesita muchas lineas de código para ser implementado. Se ha podido comprobar que para un mismo programa utiliza muchas menos que Java o C.
    3. Un programa en LUA no necesita gestionar la memoria.
    4. Es fácil de compartir dado que al no necesitar compilación, únicamente se necesita colocar el programa en el fichero de plugins de Wireshark.
  - Desventajas:
    1. En tiempo de ejecución es algo más lento que C, por lo que para programas que requieran de muchos recursos se notará la diferencia.
    2. Aún no tiene implementadas tantas funciones como otros lenguajes de programación.
    3. No se utiliza tanto como C para implementar plugins en Wireshark, por lo que es más complicado encontrar información al respecto.

  En el apartado *3. Desarrollo del plugin* se habla de cual de los disectores se escoge para el plugin y se explican las funciones necesarias para desarrollarlo.

Una vez conocido que es Wireshark como un analizador de tráfico con multitud de funcionalidades de búsqueda, podemos preguntarnos por qué decidimos crear el plugin en Wireshark y no por navegador, ya que el protocolo OpenID Connect funciona íntegramente en HTTP. Por esta razón sería lógico pensar que igualmente el mismo contenido que aparece en Wireshark queda registrado en la herramienta de analizador de tráfico que existen actualmente en los navegadores web, tales

como Firefox o Chrome, por ejemplo.

Whireshark frente a otros analizadores
Las razónes por las que escogimos Whireshark son las siguientes:

- Whireshark da la opción de descifrar el tráfico si este se envía por el puerto 443 de HTTPS a diferencia del tráfico recogido en el navegador. Esto se puede realizar únicamente en entornos de prueba o entornos desplegados por nosotros y conociendo la clave privada y pública que cifra el tráfico.

- A diferencia de el analizador de tráfico de un navegador Web, los filtros de búsqueda en Wireshark son mucho más intuitivos y eficaces.
  Por ejemplo, la **Figura 17** muestra como se observaría la Respuesta de Token realizada por nuestro servidor OpenID Connect en un navegador frente a lo que se vería en Wireshark, mostrado en la **Figura 18**:

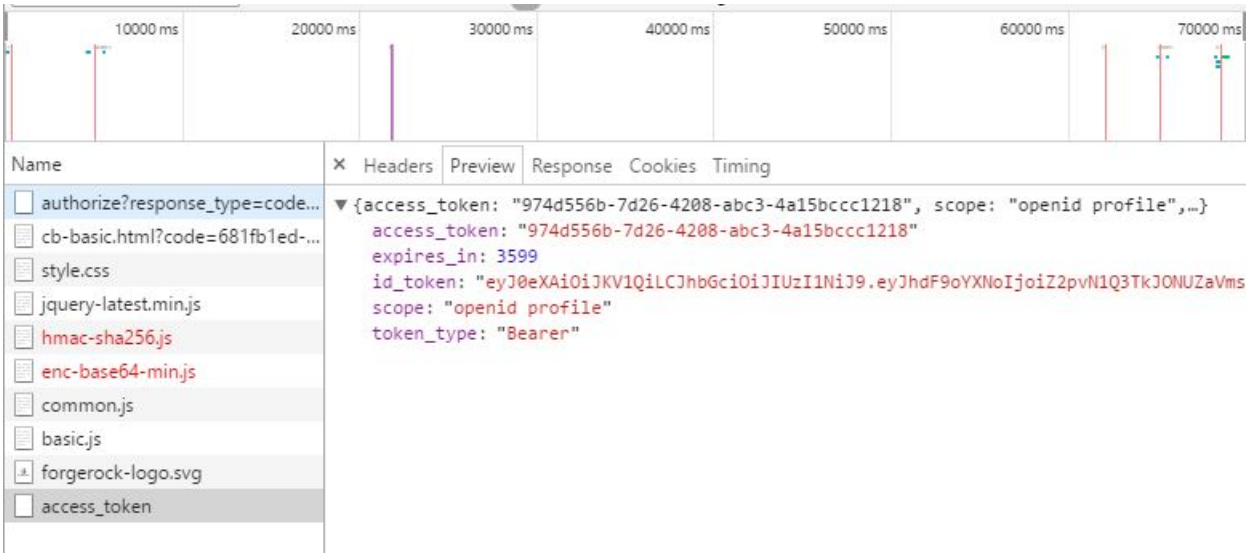

Figura 17: Respuesta de Token analizada en navegador Chrome

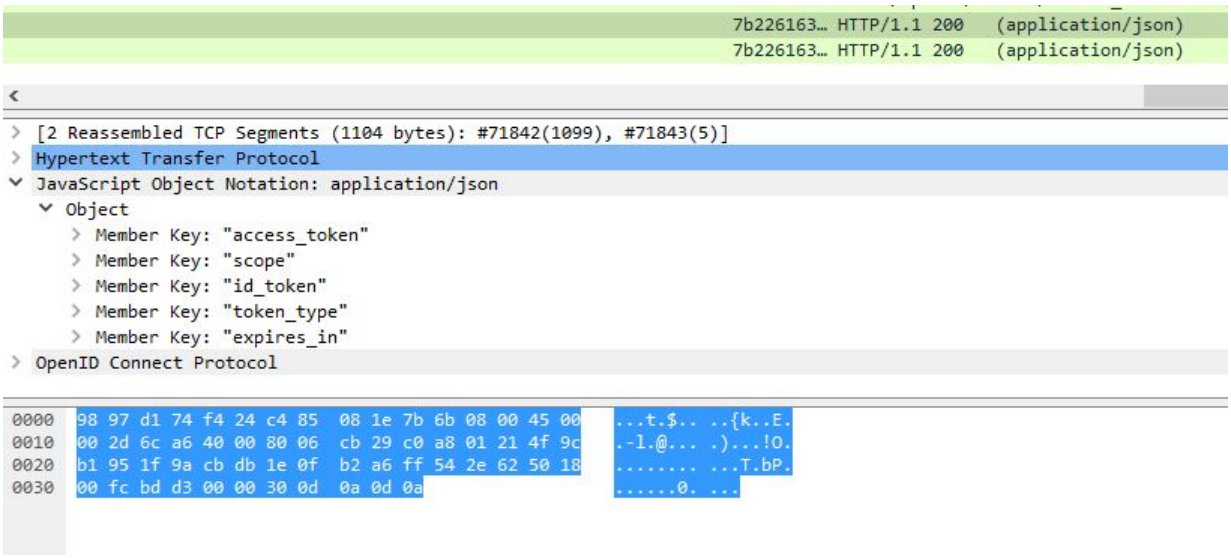

Figura 18: Respuesta de Token analizada en Whireshark

Como se puede observar Wireshark desglosa toda la información de la petición de forma más intuitiva que la herramienta de análisis de tráfico de Chrome, primero analizando la petición y sus cabeceras y luego usando un disector JavaScript Object Notation (JSON) para mostrar los detalles más ordenados.

2.3.3. Analizadores de tráfico SAML

Llegados a este punto ya está claro por qué se ha escogido un plugin programado en LUA de Wireshark, pero, ¿por qué se ha escogido OpenID Connect frente a SAML? Se ha expuesto en el apartado *2.1. Infraestructuras de gestión de identidad* ambos protocolos y se ha hablado de las ventajas de utilización de cada uno de ellos y como funcionan exactamente, pero aún sigue sin ser una razón suficiente el escoger OpenID Connect frente a SAML.
Hay una razón principal en ello, y es que debido a que SAML es un protocolo más antiguo que OpenID Connect, actualmente ya existen plugins de navegador para analizar el tráfico. Uno de ellos es el siguiente:

- **SAML Devtools Extension**
  Es un plugin diseñado para el navegador Chrome y que configura un menú extra en la herramienta de desarrollador para capturar el tráfico, como muestra la **Figura 19**:

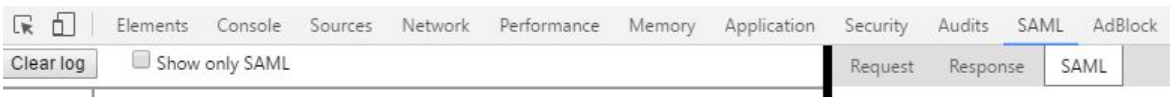

Figura 19: Menú de acceso a la Devtools Extension

Su objetivo es analizar el tráfico mientras permanezca activada, y una vez detecta que se están realizando peticiones SAML ilumina la traza además de exponer en la pestaña de SAML el XML que se ha enviado o recibido, con los detalles de la petición, como podemos ver en la **Figura 20**:

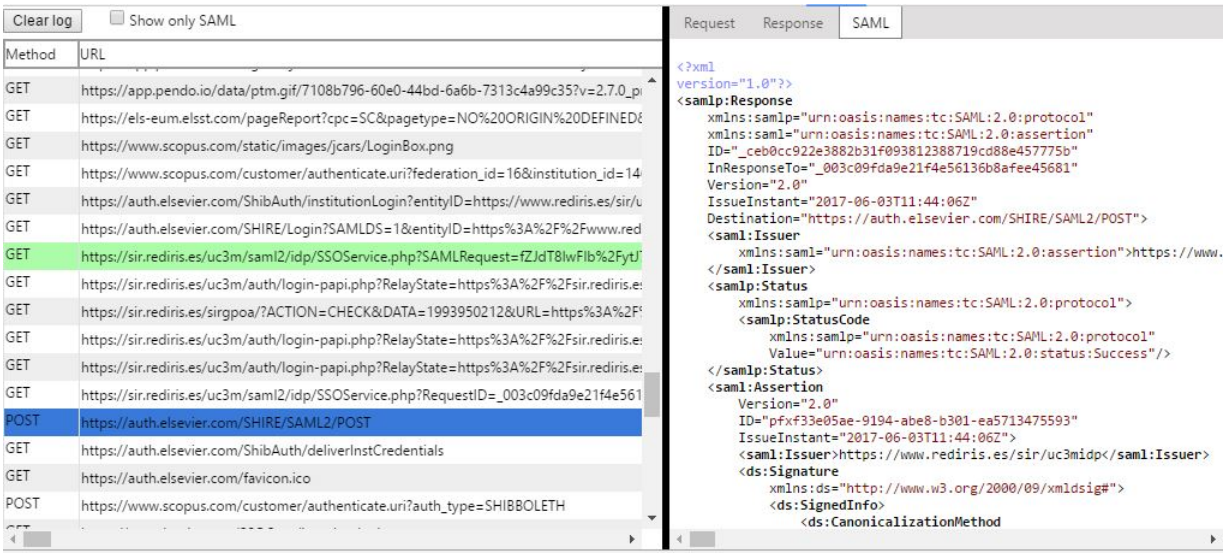

Figura 20: Trazas SAML y resultados mostrados por Devtools Extension

Se pueden encontrar más detalles sobre esta extensión en el Chrome web store[15].

2.4. Marco regulador y entorno socioeconómico

Para la creación del plugin de OpenID Connect no se ha encontrado ningún obstáculo legal en el camino que no permita la realización del plugin o que se tengan que tomar medidas de algún tipo para su desarrollo.
Todos los componentes son gratuitos y de acceso público, tanto las tecnologías utilizadas para la infraestructura base, en este caso de Wireshark y OpenAM, como del lenguaje de programación utilizado, LUA.
Si bien todo lo anterior es cierto, hace que se plantee la posibilidad de que para la mejora de la infraestructura utilizada sería posible escoger una versión no gratuita de OpenAM, ya que, como trataremos en el apartado **4. Arquitectura de despliegue** la versión tiene su respectiva limitación. Por ello, si nos planteamos conseguir la versión oficial de OpenAM podemos obtener las siguientes ventajas:

- **Versión final más estable**
  La versión gratuita instalada es una versión de pruebas que no ha pasado por todos los procedimientos de desarrollo ni tampoco ha sido probada, por lo que podría dar errores puntuales a diferencia de la versión con licencia oficial.

- **Base de datos más amplia**
  La versión de OpenAM tiene otra restricción, y es que la base de datos utilizada para el almacenamiento de los usuarios finales que vayan a autenticarse está limitada por número, haciendo que para grandes organizaciones con numerosos clientes y que pretendan tener un entorno de autenticación basado en OpenAM sea inviable y haya que recurrir a la versión final con sus licencias pagadas.

- **Funcionalidades no desarrolladas**
  Al tratarse de la última versión realizada y que no ha pasado por todos las pruebas pertinentes, algunas de sus funcionalidades directamente no han sido terminadas, por lo que o generan error o no son ni accesibles. Estas funcionalidades pueden variar de una versión a otra por lo que la versión Nightly (versión gratuita) no dan ninguna garantía de que un entorno instalado para una determinada versión vaya a funcionar exactamente igual en la siguiente.

3. Desarrollo del plugin

Teniendo ya una visión más global de qué tecnologías se van a utilizar para el desarrollo del plugin, del funcionamiento del protocolo OpenID Connect y de las tecnologías web que entran en juego para entender mejor las peticiones qué se realizan entre cliente y servidor, en este apartado finalmente se habla del desarrollo del plugin que se ha realizado y que medidas se han tomado para simplificar su funcionamiento y para que sea lo más didáctico posible, visualmente en Wireshark y con la programación del mismo.

3.1. Disectores en LUA

Se ha hecho referencia en el apartado **2.3.2. Wireshark** sobre los distintos tipos que existen de disectores hoy en día y para que se utiliza cada uno de ellos. Para el desarrollo de este apartado primero se va a comentar las principales funciones que ayudarán a crear un nuevo disector de forma genérica y que aplica a los distintos tipos, sin entrar en detalles de las funciones específicas para cada uno de ellos, teniendo en cuenta la guía de desarrollador de Wireshark[16]. Una vez conseguido se va a explicar que tipo de disector se ha escogido y las razones por las que se hizo esa elección en el apartado de **3.2.1. Tipo de disector escogido**.

3.1.1. Análisis del nuevo protocolo

Para la creación de un disector es importante primero conocer las características del protocolo que se va a implementar y hacer un análisis de los principales componentes que se considera relevantes para su desarrollo.
Una vez lo se tenga la estructura clara, en LUA existen diferentes funciones que ayudan a la creación del disector y que se pueda amoldar al protocolo.

3.1.2. Creación del nuevo protocolo

En primer lugar creamos el protocolo en sí y escogemos un nombre para él. Esto se realiza mediante la función *Proto()*, como se puede observar en este ejemplo:

trivial_proto = Proto("trivial","Trivial Protocol")

Con ello definimos que el nombre corto del protocolo será *trivial* y su nombre completo *Trivial Protocol*.

3.1.3. Función de disección

En LUA existe la función dissector() que realiza la disección del protocolo una vez es llamado, como por ejemplo:

function trivial_proto.dissector(tvb,pinfo,tree)

Esta función se llama para diseccionar los paquetes que se le presentan. Los datos de paquetes se mantienen en un buffer especial que se hace referencia en este ejemplo como *tvb*.
El segundo parámetro, pinfo, es el registro de información de los paquetes que contiene datos generales sobre el protocolo, y también sirve para poder actualizar

la información en la función.
El parámetro *tree* es el lugar donde se guardan los parámetros de la disección en detalle, es la raíz de nuestro árbol de disección y del cual colgaremos las distintas ramas que irán formándolo.

3.1.4. Configuración de los nuevos campos

Una vez creado el protocolo, se crean los campos importantes y representativos que tiene. Para ello, existen varias formas de "enganchar", por así decirlo, subárboles al árbol principal:

- Mediante la función de ProtoField podemos ir generando campos y añadiéndolos posteriormente al disector, por ejemplo:
  src_F = ProtoField.string("trivial.src","Source")
  Esta función tiene estas características dependiendo del tipo de dato que sea necesario registrar en el disector:

  1. Para números:
     ProtoField.type (abbr, [name], [desc],[base], [valuestring], [mask])
     type puede tomar uno de los siguientes valores: uint8, uint16, uint24, uint32, uint64, framenum.
  2. Para otros tipos:
     ProtoField.type (abbr, [name], [desc])
     type puede tomar uno de los siguientes valores: float, double, string, stringz, bytes, bool, ipv4, ipv6, ether, oid, guid.

  Esta forma de implementación de los campos del disector permite posteriormente realizar búsquedas y filtrar por determinadas características directamente en Wireshark.

- Creando directamente un subárbol que se alimenta del contenido del buffer sin tener ningún campo predefinido, como por ejemplo:
  local subtree = tree:add(trivial_proto,buffer(),"Trivial Protocol Data")
  Esta función la explicaremos más adelante ya que sirve para añadir contenido del buffer o el contenido de un campo que se haya creado de la anterior forma explicada.

Dependiendo de la forma en la que vayamos a implementar nuestro disector se escoge una forma u otra, aunque ambas son igualmente válidas.

3.1.5. Obtención de los datos del paquete

Al principio del apartado hablamos del buffer especial, tvb (Testy Virtual Buffer), que tiene la siguiente estructura:
Tvb ([offset], [length])
Los datos guardados en el buffer se pueden transformar en otro tipo de datos, que más concuerde con nuestro protocolo, añadiendo una de las siguientes funciones: uint, le_uint, float, le_float, ipv4, le_ipv4, ether, string, bytes.
Un ejemplo de ello sería el siguiente

local msgid_range = tvb(0,4)
local msgid = msgid_range:uint()
Con ello estamos identificando que el msgid será de tipo uint.
Cabe destacar que el uso del buffer no es necesario en los disectores de tipo Post-Disector, ya que pueden obtener información relevante de otros disectores utilizando funciones propias para esta función y los cuales se tratan en el punto **3.1.7 Obtención de los valores de los campos configurados**.

3.1.6. Concatenación de los campos configurados al árbol

Ahora que se ha explicado como se obtienen los datos del buffer, podemos explicar como añadir valores al árbol de disección, modificando alguno ya existente o bien creando un subárbol directamente:

- Podemos añadir un elemento al árbol con la función *add()*
  treeitem:add ([field — proto], [tvbrange], [label])
  Como se puede observar podemos añadir un campo creado previamente o bien el buffer. Un par de ejemplos para esta función serían los siguientes:

  1. *local subtree = tree:add(tvb(0,2),"The first two bytes: " .. buffer(0,2):uint())*
  2. *local subtree = tree:add(trivial_proto,"Trivial Protocol Data")*

  En el primer ejemplo estamos añadiendo directamente el buffer al árbol principal y la función devuelve un hijo al cual podemos concatenarle más hijos. En el segundo ejemplo estamos añadiendo el campo creado con anterioridad en el disector, llamado trivial_proto.

- También podemos modificar un subárbol ya existente con las siguientes funciones:

  - treeitem:set_text (text)
  - treeitem:append_text (text)
  - treeitem:add_expert_info ([group], [severity], [text])
  - treeitem:set_generated ()

3.1.7. Obtención de los valores de los campos configurados

Una función muy útil que tienen los disectores es la función Field.new(). Gracias a ella podemos recuperar el valor de cualquier otro disector para poder utilizarlo en el nuestro propio. Un ejemplo de ello sería el siguiente:

```
ip_src_f = Field.new("ip.src")
trivial_proto = Proto("trivial","Trivial Postdissector")
function trivial_proto.dissector(buffer,pinfo,tree)
local ip_src = ip_src_f()
    « resto del contenido del disector »
end
```

Podemos observar en el ejemplo que podemos utilizar cualquier otro campo predefinido en otro protocolo aunque no sea el nuestro. En este caso se ha utilizado el campo de IP Origen del disector IP implementado en Wireshark.
Primero se guarda el contenido del campo fuera de la función de disección de nuestro disector, y posteriormente se crea una variable local que guarda el contenido del valor.

3.1.8. Modificación de las columnas de Wireshark

En Wireshark existen columnas predefinidas donde se puede estructurar la información de los distintos disectores, como se muestra en la **Figura 21**:

Protocol	Length	Full request URI	Data	Info
TCP	54			63567 → 80 [RST, ACK] Seq=3 Ack=2 Win=0 Len=0
TCP	60			80 → 63567 [RST] Seq=2 Win=0 Len=0
HTTP	938	http://pruebasopam.hop…		GET /openam/json/serverinfo/* HTTP/1.1
TCP	773			[TCP segment of a reassembled PDU]
HTTP	59		7b22646f…	HTTP/1.1 200 (application/json)
TCP	60			54086 → 8090 [ACK] Seq=4316 Ack=5550 Win=15872 Len=0
TCP	60			80 → 64264 [FIN, ACK] Seq=927 Ack=1125 Win=32128 Len=0
TCP	54			64264 → 80 [ACK] Seq=1125 Ack=928 Win=524544 Len=0
TCP	54			64264 → 80 [FIN, ACK] Seq=1125 Ack=928 Win=524544 Len=0
TCP	54			64264 → 80 [RST, ACK] Seq=1126 Ack=928 Win=0 Len=0
TCP	60			80 → 64264 [ACK] Seq=928 Ack=1126 Win=32128 Len=0
TCP	1179			[TCP segment of a reassembled PDU]

Figura 21: Información de las columnas en Wireshark

Principalmente daremos uso a dos funciones que permitirán modificar el contenido de las columnas y poder mostrar con nuestro disector la informacion que creamos más relevante:

- pinfo.cols.protocol: permite la modificación de la columna con nombre *Protocol*, que se puede observar en la imagen, y con poder añadir el nombre de nuestro propio protocolo, como en el siguiente ejemplo:
  pinfo.cols.protocol = trivial_proto.name

- pinfo.cols.info: permite la modificación de la columna con nombre *Info* para que añada alguna característica que consideremos importante a la hora de analizar nuestro protocolo. Un ejemplo de ello sería el siguiente:
  pinfo.cols.info = ("Message Id: " .. buffer(2,1):uint())

En realidad prácticamente todas las columnas son modificables, pero estas dos funciones son las más relevantes a la hora reflejar los valores de nuestro disector.

Existen más funciones que permiten la completa personalización de nuestro disector, pero el desarrollo se centra en los más relevantes y sin los cuales no se podría llevar a cabo la disección. El resto de funciones se pueden encontrar en multitud de guías que existen actualmente en la web principal de Wireshark.

3.2. Medidas tomadas para la configuración del plugin

Una vez explicado como funcionan los disectores y cómo se configuran correctamente, es hora de explicar la creación de nuestro plugin y las medidas

que se han tomado para su programación. Para ello nos centraremos en cuatro apartados para la comprensión del plugin:

1. Tipo de disector escogido
2. Configuración del plugin
3. Diferenciación por tipo de flujo
4. Diferenciación por tipo de petición

Con estas pautas abarcaremos todos los pasos tomados a la hora de la implementación el plugin para reflejar las decisiones de la forma más ordenada posible.

3.2.1. Tipo de disector escogido

Nos gustaría empezar hablando de la elección que se ha realizado acerca del tipo de nuestro disector.
Se ha hablado anteriormente en el apartado **2.3.2. Wireshark** de los distintos disectores que existen hoy en día y qué caracterizaba a cada uno, por ello se van a exponer las razones que se han seguido para la elección del disector:

1. Analizando el protocolo de OpenID Connect se puede observar que utiliza distintos tipos de disectores definidos en Wireshark:

   - HTTP
   - application/json
   - application/x-www-form-urlencoded

   Por esta razón, se descarta el utilizar un disector de tipo *Listener* ya que este tipo de disector es llamado después de que se utilice un disector en específico. Al utilizar tantos tipos de disectores para analizar los datos de los paquetes transmitidos entre los distintos actores que entran en juego en OpenID Connect, es poco útil este tipo de disector porque habría que configurar en un único programa todos los disectores relevantes y en vez de tener un único disector acabaríamos teniendo tres con distintos nombres.

2. Otra característica del protocolo OpenID Connect es que utiliza protocolos que ya están correctamente configurados en Wireshark y en los cuales podemos basar para configurar los propios campos en base a ellos. En consecuencia, descartamos la utilización del tipo *disector* que es el disector genérico que se utiliza cuando no podemos basarnos en otro disector generado y del cual se extrae toda la información útil de los paquetes de datos.

3. Interesa analizar todos los paquetes que se intercambian entre los distintos actores, para poder comprobar si reúnen cada uno de ellos las características que ofrece OpenID Connect.

Este razonamiento deja únicamente la opción de *Post-Disector*, que es la opción más acertada y correcta para configurar nuestro plugin que analizará las trazas enviadas utilizando el protocolo OpenID Connect.

3.2.2. Configuración del plugin

En este apartado tratamos la configuración del plugin que se ha realizado siguiendo los mismos pasos que en el apartado **3.1. Disectores en LUA** para seguir la misma estructura y que se pueda comprobar que únicamente siguiendo esos pasos podemos tener un disector en Wireshark perfectamente configurado y útil; añadiendo un apartado al final sobre las medidas que hay que tomar para la configuración de un Post-Disector.

1. **Análisis del nuevo protocolo**
   Para la creación más didáctica posible del plugin se ha visto necesario la configuración de unos campos donde describiesen en cada uno de los paquetes de OpenID Connect qué tipo de flujo seguía la petición y qué tipo de petición era.
   Además, consideramos relevante que reflejase los parámetros que hicieran que esa petición estuviese catalogada en un determinado flujo y también los parámetros que reflejaban el tipo de petición que era. Por ello y después de analizar en profundidad el protocolo, llegamos a la conclusión de que queríamos que los campos de nuestro disector siguieran la siguiente estructura de forma general:

   a) *HTTP Request Content (longitud del contenido)*: en este apartado mostraremos el contenido más representativo de la petición y que se ha extraído directamente de un disector anterior, o bien de HTTP, TCP o Datos, dependiendo del tipo de petición.

   b) *Flow Type*: en esta sección concatenaremos al campo el nombre del flujo que está siguiendo la petición en el protocolo OpenID Connect.

   c) *Petition Type*: aquí especificaremos que tipo de petición se está realizando.

   d) *Request Parameters (número de parámetros)*: desglosaremos en este apartado los parámetros más relevantes de la petición para que se pueda observar en todo momento por qué se ha escogido cada uno de los tipos.

   Estos serán los campos que de forma general formarán parte de nuestro protocolo. Más adelante se especificará si en algún tipo de petición se hará hincapié en algún otro detalle, cuando hablemos de las medidas tomadas para cada una de las peticiones.

2. **Creación del nuevo protocolo**
   Para la creación del nuevo protocolo únicamente se necesita la función *proto()*, que se ha configurado de la siguiente manera, como se muestra en la **Figura 22**:

```
-- Creamos un nuevo protocolo, pero no lo registramos aun
local OIDC_proto = Proto("oidc","OpenID Connect Protocol")
```

Figura 22: Creación del protocolo OIDC

Se ha escogido como nombre corto *oidc* y como nombre largo descriptivo *OpenID Connect Protocol*

3. **Función de disección** Para la configuración de la función de disección se necesita crear la función dissector(tvb,pinfo,tree) y empezar a desarrollar dentro las especificaciones de nuestro disector, como se muestra en la **Figura 23**:

```
-- The dissector function
function OIDC_proto.dissector (tvb, pinfo, tree)
```

Figura 23: Creación de la función de disección de OIDC

4. **Configuración de los nuevos campos**
En el apartado de análisis del protocolo ya expusimos de forma general los campos que considerábamos importantes a la hora de la realización de nuestro protocolo.
Después de realizar más investigaciones para exponer la mejor forma de estructurar el protocolo OpenID Connect abarcando la totalidad de tipos de peticiones y de flujos, se han llegado a la conclusión de que serían necesarios los siguientes campos, como se muestra en la **Figura 24**:

```
-- Primero creamos los campos principales del disector
local f_req = ProtoField.string("oidc.content", "HTTP Request Content")
local f_flow = ProtoField.string("oidc.flow", "Flow type")
local f_petition = ProtoField.string("oidc.petition", "Petition type")

-- Creamos los campos del arbol del ID Token para las Token Response
local f_id_token = ProtoField.string("oidc.id_token", "Claim id_token")
local f_header = ProtoField.string("oidc.id_token.header", "JOSE Header")
local f_payload = ProtoField.string("oidc.id_token.payload", "Payload")
local f_signature = ProtoField.string("oidc.id_token.signature", "Signature")

-- Creamos los campos del árbol de parámetros
local f_claims = ProtoField.string("oidc.claim", "Claim")

-- Añadimos los campos al protocolo OIDC
OIDC_proto.fields = {f_req, f_flow, f_petition, f_parameters, f_id_token, f_header,
                     f_payload, f_signature, f_claims}
```

Figura 24: Campos configurados del protocolo OIDC

Se puede observar en la **Figura 24** que se han creado tres grupos de campos que difieren unos entre otros según su colocación en el árbol y todos ellos, a excepción de uno se han creado usando la función ProtoField.string():

- El primer grupo está compuesto por los campos que aparecerán de forma fija en todas las trazas enviadas entre cliente y servidor indistintamente del tipo que sean, y que cuelgan directamente del árbol principal. Están formados por:

- **Campo de contenido:** se mostrará la información más relevante del contenido de la petición y que será diferente dependiendo del tipo de petición que sea. Más adelante, en el apartado **3.2.5 Diferenciación por tipo de petición** se describe el contenido el cual se trata.
- **Campo de flujo:** en este campo se mostrará el tipo de flujo que está siguiendo la autenticación según las especificaciones de OpenID Connect.
- **Campo de petición:** se expondrá en este campo el tipo de petición que se está realizando según las especificaciones de OpenID Connect.
- **Campo de parámetros:** este campo sirve para recoger todos los *claims* que se han utilizado en la petición, concatenándolos como subárboles de este mismo. Como en sí no resulta útil para realizar búsquedas en Wireshark, se ha decidido no crearlo como un campo utilizando la función ProtoField.type().

Un ejemplo de cómo se muestran estos campos en Wireshark sería el mostrado en la **Figura 25**:

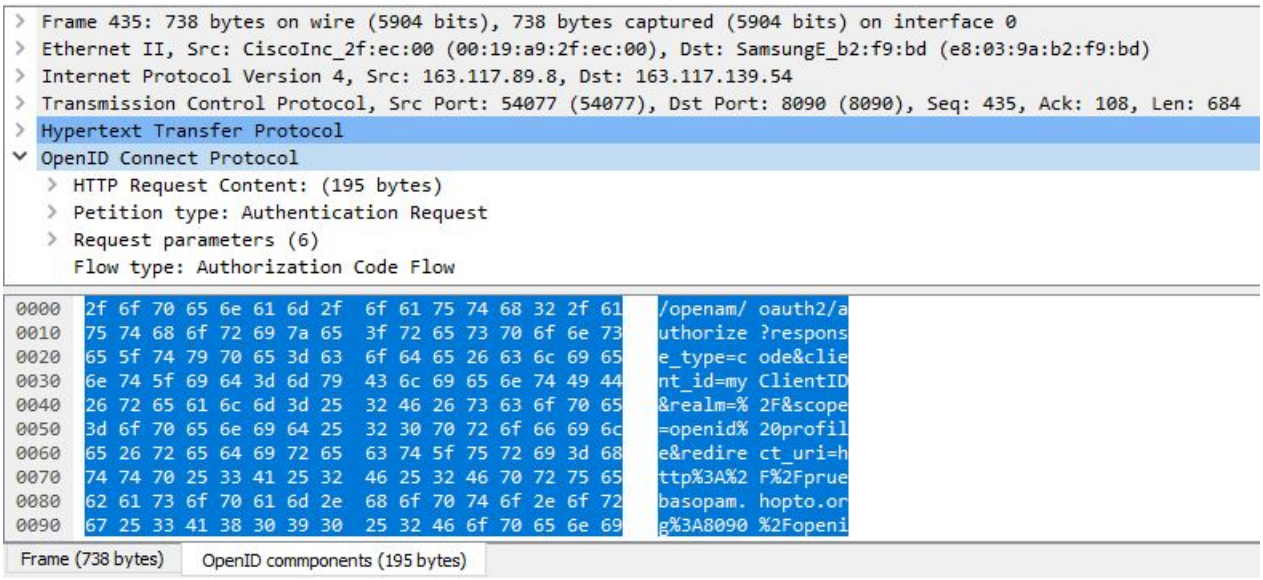

Figura 25: Campos principales de OIDC en la interfaz de Wireshark

- El segundo grupo está compuesto por los campos que cuelgan de la rama de parámetros cuando se detecta que es una respuesta de Token. Se puede observar también que el *claim id_token* se trata de forma distinta a los demás y se le ha dado un nombre propio al tratarse de un *claim* importante y que va codificado. Este subárbol está compuesto por los siguientes campos:
  - **Campo de *ID Token*:** este campo será el único hijo del árbol de parámetros de todos los que aparecen en este grupo

de campos. Está configurado para que muestre los datos del *ID Token* sin descodificar y que cuelguen en él el resto de campos ya descodificados.

- **Campo de la cabecera *ID Token*:** este campo mostrará la información de la cabecera JOSE del *ID Token*.
- **Campo de *payload* de *ID Token*:** este campo mostrará los valores del *payload*, guardándolos en subárboles que penderán de él.
- **Campo de firma de *ID Token*:** este campo guardará la información de la firma del *ID Token*.

El resultado de estos campos en la interfaz de Wireshark se puede ver mejor en la **Figura 26**:

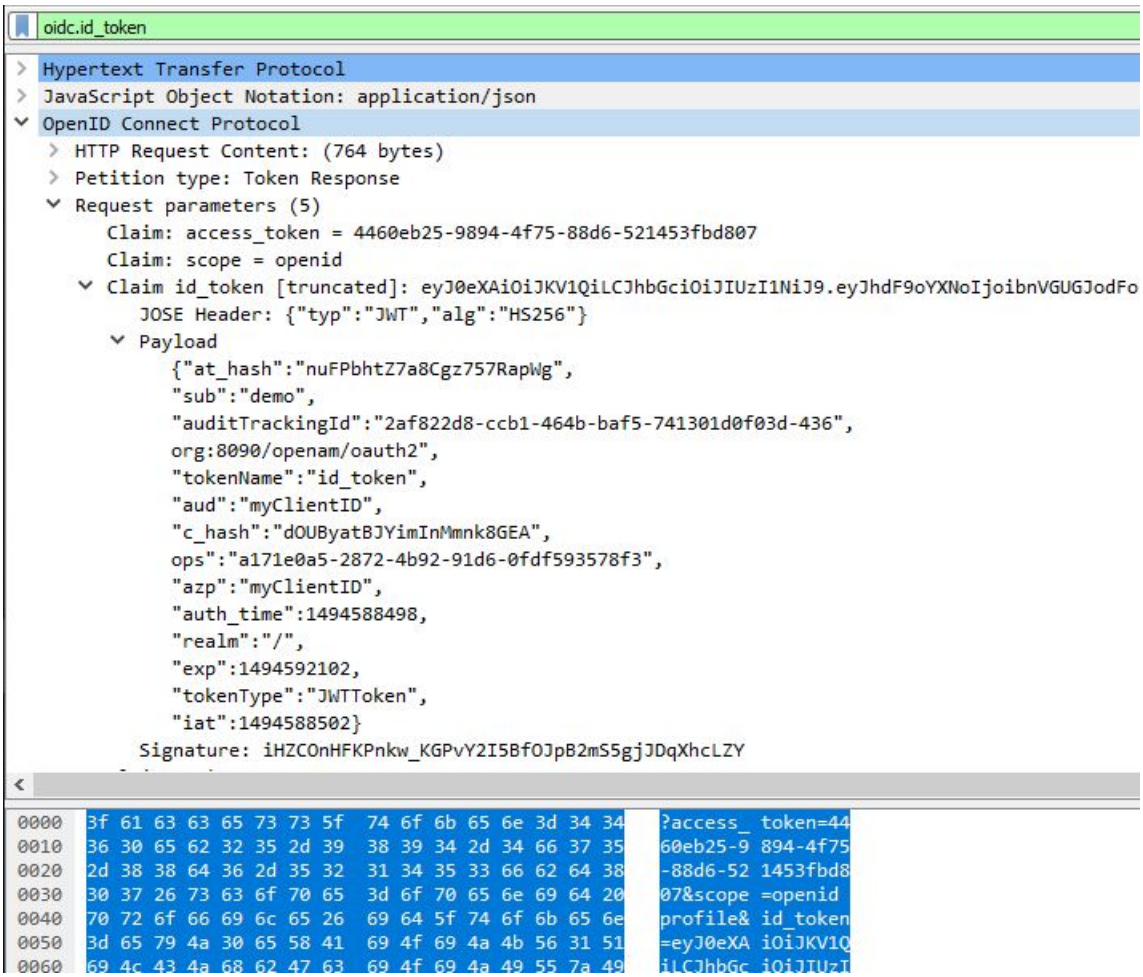

Figura 26: Campos de *ID Token* de OIDC en la interfaz de Wireshark

- El tercer grupo de campos está formado por un único tipo de campo, que son los hijos comunes que cuelgan del campo de parámetros y que lo forman todos lo *claims* que aparecen en la petición.
  Todos tienen el mismo formato, a excepción del campo *ID Token* como

se ha explicado en el anterior punto.
Un ejemplo de como se ven este tipo de campos es el que se muestra en la **Figura 27**:

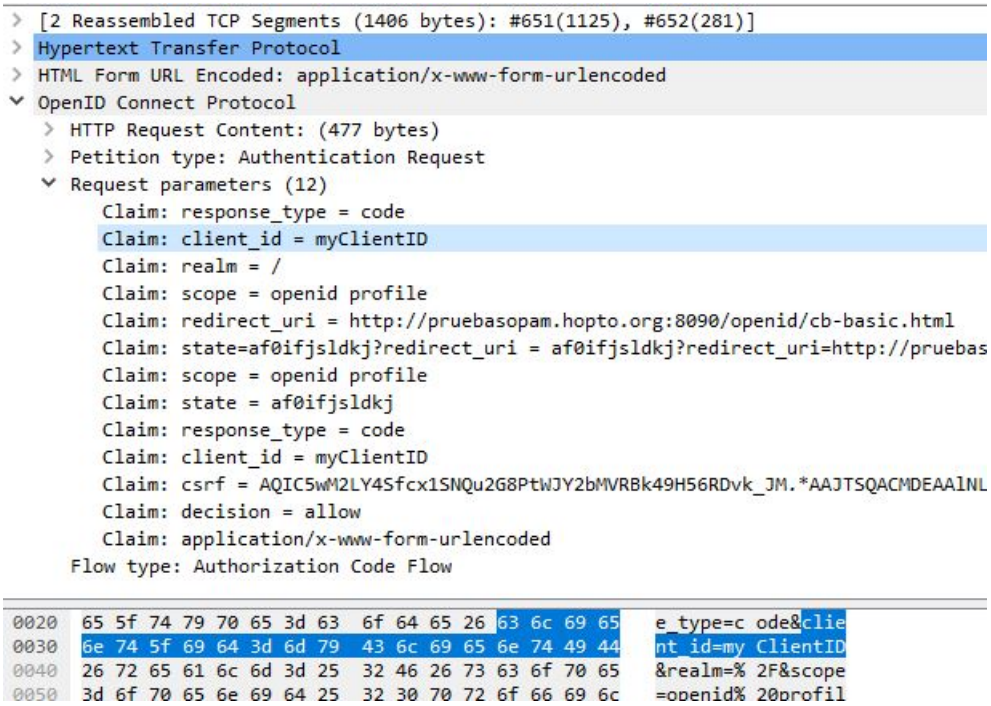

Figura 27: Campos de *claims* de OIDC en la interfaz de Wireshark

Estos han sido todos los campos utilizados en nuestro plugin y que mostraremos más adelante cómo los configuramos, para que, además de concatenar datos, muestre alguna información relevante extra.

5. **Obtención de los datos del paquete**
Como ya hicimos referencia en el apartado **3.1.5. Obtención de los datos del paquete**, en el caso de los Post-Disectores no necesitan obtener información del buffer ya que extraen directamente los datos de los campos configurados. Igualmente, para realizar una concatenación en el árbol lo ideal sería utilizar la siguiente función para convertirlo en un buffer parecido al tvb y poder manejarlo:

```
local tab = ByteArray.new(string2hex(content)):tvb("OpenID commponents")
local tvb_range = tab()
```

Figura 28: Obtención de los datos del paquete en OIDC

Estas sentencias convierten a la variable *tvb_range* en una variable userdata con lo que se mostrará la información en una pestaña de Wireshark de la siguiente manera:

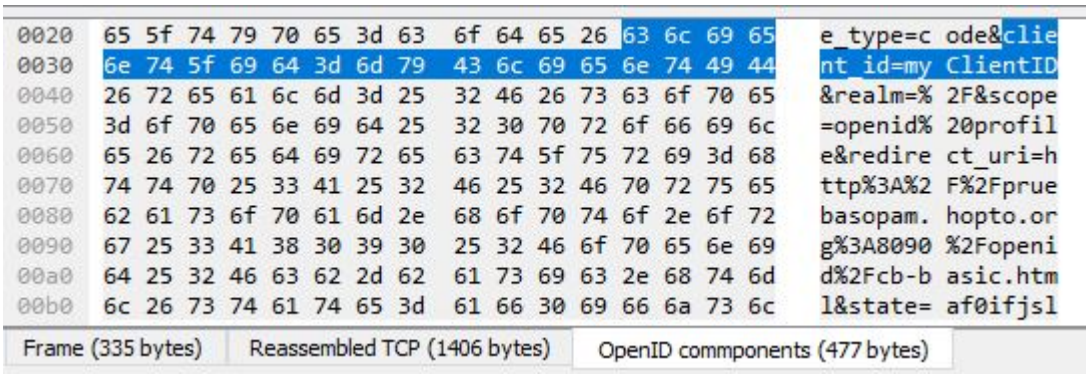

Figura 29: Pestaña de datos de OIDC en Wireshark

6. **Concatenación de los campos configurados al árbol**
   Una vez conocemos todos los campos que se necesita y la forma en la que se va a programar nuestro protocolo OIDC para que escoja la información que requerimos para analizar, es momento de agregar el contenido al árbol principal. En la **Figura 30**se puede observar todas las funciones de concatenación de subárboles que tenemos configuradas en el Post-Disector, omitiendo el resto de código:

```
-- Añadimos el protocolo al árbol
local subtree = tree:add(OIDC_proto, tvb_range)
...
-- Añadimos los datos del contenido relevante al árbol
subtree:add(f_req, tvb_range, "(" .. tvb_range:len() .. " bytes)"):add(tvb_range, content)
...
--Una vez extraemos la información sobre el tipo de petición que es, creamos el subárbol de peticiones
-- y lo añadimos al árbol
local petition_tree = subtree:add(f_petition, tvb_range(0,0), valor)
petition_tree:add(tab(), "Content:  ".. unescape(content))
...
-- Si la petición es Authenticatión Response o un Token Request entonces se añade además la raíz URI:
-- En el caso de la Authenticatión Request sería "authorize" y del Token Request "access_token"
-- (Este cliente no sigue las especificaciones que indica que tendrían que ser "Token")
if pet == petition.AUTHENTICATION_REQUEST or pet == petition.TOKEN_REQUEST then
    -- Añade al arbol el contenido de la raíz en el caso de que sea una
    petition_tree:add(tab(), "URI root: /".. unescape(req))
    -- Si la petición es Authenticatión Response o un Token Request entonces se
elseif pet == petition.AUTHENTICATION_RESPONSE then
    ...
    petition_tree:add(tab(), "Code = ".. unescape(sub))
    ...
end
...
-- Añadimos el subárbol de parámetros
local parameters_tree = subtree:add(tvb_range, "Request parameters")
...
-- En el caso de contener el id_token entonces guardamos todos las partes que lo componen decodificándolas antes
id_tree = parameters_tree:add(f_id_token,tab(si-1, xlen), decode_v)
...
id_tree:add(f_header, tab(plen, ylen), dec(parte1))
...
payload_tree = id_tree:add(tab(plen, ylen), "Payload")
...
payload_tree:add(tab(si-1, xlen), unescape(par_id))
...
id_tree:add(f_signature, tab(plen, ylen), parte3)
...
-- Añadimos el resto de los claims de las peticiones
parameters_tree:add(f_claims,tab(si-1, xlen), decode_p.. " = ".. decode_v)
...
-- Finalmente agregamos el tipo de flujo que es al árbol principal, utilizando una función antes para ello y el
-- número de parámetros que se han analizado en la petición
local flow_tree = subtree:add(f_flow, tvb_range(0,0), tipo)
parameters_tree:append_text(" (" .. count .. ")")
```

Figura 30: concatenación de los campos configurados en OIDC

Se puede observar que al ir configuradas exactamente en este orden reflejan también el mismo orden que en la interfaz de Wireshark, además podemos también comprobar los datos extra que aparecen en la petición y que no programamos en el momento de definir los campos.
Eso es gracia a las funciones de *treeItem:append_text* y a la concatenación en LUA que facilita el trabajo con un código tan simple.

7. **Obtención de los valores de los campos configurados**
   Los campos configurados de otros disectores que se han utilizando en nuestro código son los mostrados en la **Figura 31**:

```
local f_udp = Field.new("udp.dstport")
local f_http_uri = Field.new("http.request.uri")
local f_http_connection = Field.new("http.connection")
local f_http_location = Field.new("http.location")
local f_http_response_code = Field.new("http.response.code")
local f_http_request_method = Field.new("http.request.method")
local f_data_type = Field.new("data.data")
local f_tcp_data = Field.new("tcp.segment_data")

-- La función de disección
function OIDC_proto.dissector (tvb, pinfo, tree)

    local http_uri = f_http_uri()
    local http_connection = f_http_connection()
    local http_location = f_http_location()
    local http_response_code = f_http_response_code()
    local udp_port = f_udp()
    local http_request_method = f_http_request_method()
    local tcp_data = f_tcp_data()
    local data_type = f_data_type()
```

Figura 31: Concatenación de los campos configurados en OIDC

Tal como explicamos en el apartado **3.1.7. Obtención de los valores de los campos configurados** primero se llaman fuera de la función y luego se definen localmente para poder utilizarlos adecuadamente.
Se han utilizado numerosos protocolos pero no todos los campos han sido utilizados de la misma manera. Podemos desglosar su funcionalidad de la siguiente forma:

- Comprobación en funciones: este tipo de campos sirven en el código para comprobar el estado de la petición.
  Se tendría en cuenta en este grupo los campos con nombre:
  - *udp_port*: utilizamos este campo para asegurarnos de que si tiene activo en la petición el campo de puerto UDP entonces cerramos la conexión y no se seguirá tratando ni procesando, aún cuando se trate de una petición HTTP.
  - *http_coonection*: al principio del código comprobamos si es una petición HTTP y en caso contrario se sale del disector.
  - *http_response_code*: la utilizamos para conocer el tipo de código de respuesta HTTP que tiene la petición.
    En caso de coincidir con el código 302, que es un redirect en HTTP, entonces tendríamos que guardar como contenido relevante de la petición los datos que aparecen en *Location*.
    En cambio si el código es de tipo 200, que es el código HTTP de OK, los datos vendrían en formato application/json según las especificaciones, por lo que habría que recoger los datos guardados en él y tratarlos posteriormente.

- *http_request_method*: este campo muestra si el método de petición es mediante un POST o un GET. Este dato es muy importante, ya que como explicamos en el apartado **2.1.1. OpenID Connect** si el método utilizado es un GET la información estará concatenada en la URI, en cambio si es un POST habría que recogerla dependiendo del contenido mismo de la petición.

- Analizar los datos contenidos: estos campos los utilizaremos porque contienen información relevante a cerca de la petición y los utilizaremos para realizar un análisis posterior para conocer el tipo de petición OpenID Connect son y que tipo de flujo están siguiendo.
  Se tiene en cuenta en este grupo los campos con nombre:
  - *http_request_uri*: como el nombre bien indica es el campo donde se guarda el contenido de la petición URI. En la función de disección una vez comprobamos si esa petición es de autenticación, verificamos si contiene el campo de petición URI contiene datos y si no es así se comprueba que estén guardados en el campo de *Location* de HTTP.
  - *http_location*: es el campo donde se guarda la URL en las peticiones POST.
  - *data_type*: este campo ya no es tan intuitivo como los anteriores. Este campo contiene los datos del disector *Data*, y es de donde sacamos los datos en bruto del contenido JSON.
    Se ha optado por sacar los datos directamente de ahí en lugar del disector JSON ya que al estar tratados previamente añade campos y nombres que no necesitábamos y que interferían en la estética que buscábamos del plugin.
    Antes de guardar su contenido en nuestro disector, al tratarse de un campo que está en Hexadecimal, primero lo transofmamos a cadena de caracteres mediante una función y luego adaptamos su contenido para que concuerde con el resto del código.
  - *tcp_data*: este campo contiene el segmento Data del disector TCP. Lo utilizamos ya que es ahí donde está contenido el tipo media con nombre application/x-www-form-urlencoded y al igual que pasaba con el contenido de tipo JSON, decidimos coger los datos en bruto del disector principal que lo guarda convirtiéndolo de Hexadecimal a cadena de texto.

Estos son todos los campos configurados utilizados, y como dijimos al inicio son bastante dispares uno de otros y al final, como se ha explicado, era necesario obtener los datos de muchos disectores distintos lo que afianza la elección de un Post-Disector.

8. **Modificación de las columnas de Wireshark**
   En nuestro plugin, referente a la modificación de columnas en Wireshark, sólo vimos necesario la modificación de la columna de protocolos:

```
-- Modificamos el nombre de la columna de protocolos para que aparezca OIDC
pinfo.cols.protocol = OIDC_proto.name
```

Figura 32: Modificación de columnas en OIDC

No se ha querido modificar las demás columnas, ya que la otra relevante que es la columna de *Info* preferimos que mantuviera el formato de *HTTP* y mostrara las peticiones con el código.
Por lo tanto, con la anterior modificación, las columnas de nuestro disector se mostrarían como se observa en la **Figura 33**:

No.	Time	Source	Destination	Protocol	Length	Info
455	34.701037	163.117.89.8	163.117.139.54	HTTP	1367	POST /openam/json/realms/roo
567	41.486632	163.117.89.8	163.117.139.54	HTTP	753	POST /openam/json/realms/roo
445	34.429836	163.117.89.8	163.117.139.54	HTTP	941	GET /openam/json/realms/root
591	41.835218	163.117.89.8	163.117.139.54	HTTP	1128	GET /openam/json/realms/root
578	41.759509	163.117.89.8	163.117.139.54	HTTP	1208	POST /openam/json/realms/roo
613	42.275206	163.117.89.8	163.117.139.54	HTTP	938	GET /openam/json/serverinfo/
584	41.774194	163.117.89.8	163.117.139.54	HTTP	1311	POST /openam/json/sessions?_
449	34.475829	163.117.89.8	163.117.139.54	HTTP	1007	POST /openam/json/users?_act
676	45.390860	163.117.89.8	163.117.139.54	OIDC	215	POST /openam/oauth2/access_t
435	34.024521	163.117.89.8	163.117.139.54	OIDC	738	GET /openam/oauth2/authorize
595	41.896459	163.117.89.8	163.117.139.54	OIDC	1154	GET /openam/oauth2/authorize
652	45.181599	163.117.89.8	163.117.139.54	OIDC	335	POST /openam/oauth2/authoriz
332	26.117097	163.117.89.8	163.117.139.54	HTTP	488	GET /openid HTTP/1.1
655	45.232631	163.117.89.8	163.117.139.54	OIDC	1059	GET /openid/cb-basic.html?co
760	50.991108	163.117.139.54	77.234.44.35	HTTP	356	GET /R/A3gKIGVjNzgyOTE2NTVmN

Figura 33: Columnas de OIDC en Wireshark

9. **Funciones de un Post-Disector**
   El Post-Disector es un tipo de disector que, como se ha visto reflejado en los anteriores pasos, se comporta igual que los demás disectores a excepción de los detalles mencionado anteriormente en otros apartados y de que es un tipo de disector que necesita ser registrado en el proceso de disección.
   Existe una función que hay que llamar para que el Post-Disector pueda ser llamado después de cada uno de los paquetes y es la siguiente:

```
register_postdissector(OIDC_proto)
```

Figura 34: Función de registro de protocolo en OIDC

Sin esa función el disector no aparecería en Wireshark así que es muy importante el utilizarlo o bien, al final del código o definirlo al principio, pero no dentro de la función de disección.

3.2.3. Diferenciación por tipo de flujo

Para que el disector pudiera diferenciar entre sí las peticiones seguían un flujo determinado u otro, se han creado las siguientes estructuras que recogen los parámetros más importantes de cada una de las peticiones con relación al flujo

para poder compararlas posteriormente en una petición que clasifica por tipo de flujo, como se muestra en la **Figura 35**:

```
local c_auth_request = {
    ["flow"] = "Authorization Code Flow",
    ["name"] = "Authentication Request",
    ["scope"] = "openid",
    ["response_type"]  = "code",
    ["factores"] = 2,
}

local i_auth_request = {
    ["flow"] = "Authorization Implicit Flow",
    ["name"] = "Authentication Request",
    ["scope"] = "openid",
    ["response_type"]  = "id_token token,id_token",
    ["factores"] = 2,
}

local c_auth_response = {
    ["flow"] = "Authorization Code Flow",
    ["name"] = "Authentication Response",
    ["scope"] = "openid",
    ["factores"] = 1,
}

local i_auth_response = {
    ["flow"] = "Authorization Implicit Flow",
    ["name"] = "Authentication Response",
    ["scope"] = "openid",
    ["response_type"]  = "id_token token,id_token",
    ["factores"] = 2,
}

local c_token_request = {
    ["flow"] = "Authorization Code Flow",
    ["name"] = "Token Request",
    ["grant_type"]  = "authorization_code",
    ["factores"] = 1,
}
local c_token_response = {
    ["flow"] = "Authorization Code Flow",
    ["name"] = "Token Response",
    ["scope"] = "openid",
    ["token_type"] = "Bearer",
    ["factores"] = 2,
}
```

Figura 35: Estructuras de las variables en OIDC

Como se puede observar únicamente se han tenido en cuenta el Flujo Código y el Flujo Implícito ya que son los únicos que soporta el cliente OpenAM, por lo que para diferenciar unas variables de otras se han incluido la letra *c* si se trata de un Flujo Código o una *i* si se trata de Flujo Implícito.
También se puede observar que para la Petición de Token y la Respuesta de Token no tienen variables locales de tipo Flújo Implícito dado que este flujo no contempla estas dos peticiones.
Mediante una función se va comprobando uno por uno todos los parámetros para ver que está cumpliendo con los obligatorios, como se puede comprobar en la

Figura 36 para el caso de la Petición de Autenticación:

```
---- Averigua si es un flujo de OpenID Connect y cual en concreto
function flow_type(parametro, valor, peticion)
    local val = nil
    local aux = "OTHER"
    local c2 = 0
    local i2 = 0
    local code = false

    if peticion == petition.AUTHENTICATION_REQUEST then
        for k, v in pairs(parametro) do
            -- Comprobación de los datos con los valores de la request de Authentication Code Flow
            for key, val in pairs(c_auth_request) do
                if string.find(v,key) then
                    if string.find(valor[k], val)  then
                        c2 = c2 + 1
                    end
                -- Comprobamos si ya ha hecho la comparación con los parámetros importantes
                elseif c2 == c_auth_request["factores"] then
                    aux = c_auth_request["flow"]
                    return val, aux
                end
            end
            for key, val in pairs(i_auth_request) do
                if string.find(v,key) then
                -- Comprobación de los datos con los valores de la request de Authentication Implicit Flow
                    if string.find(valor[k], val) or string.find(val, valor[k])  then
                        i2 = i2 + 1
                    end
                -- Comprobamos si ya ha hecho la comparación con los parámetros importantes
                elseif i2 == i_auth_request["factores"] then
                    aux = i_auth_request["flow"]
                    return val, aux
                end
            end
        end
    elseif peticion == petition.AUTHENTICATION_RESPONSE then
        for k, v in pairs(parametro) do
            if string.find(v, "code") then code = true end
            for key, val in pairs(c_auth_response) do
                if string.find(v,key) then
```

Figura 36: Función de catalogación de flujo en OIDC

Todas las peticiones son comprobadas con la misma estructura, a excepción de la Petición de Token y la Respuesta de Token ya que por el contenido que guardan y dependiendo de cómo se ha guardado la estructura se les ha añadido una cláusula que les permite añadir el flujo, el único que tienen, aunque no sea exactamente igual el contenido.

3.2.4. Diferenciación por tipo de petición

Uno de los objetivos de nuestro disector es que diferencie el tipo de petición de cada uno en el caso de pertenecer al protocolo de OpenID Connect.
Esto se ha realizado mediante dos pasos importantes, primero intentando conocer qué tiene contenido cada uno de las peticiones y que método utiliza, lo cual se detalla en el apartado **3.2.2. Configuración del plugin**, y por otro lado crear una función que termina de catalogar cada una de las peticiones una vez se comprende y se guarda adecuadamente el contenido.
En en la **Figura 37** se puede ver como está configurada la elección de contenido relevante dependiendo de la petición HTTP:

```
if http_response_code then
    method = http_response_code.value
    if http_response_code.value == 302 then
        content = http_location.value
        -- Comprobamos si no se trata de un Autentication Response del Implicit Flow.
        if string.find(content, "access_token=") then
            access = true
        end
    elseif http_response_code.value == 200 then
        -- Convertimos el JSON que está en hexadecimal a String y le cambiamos el formato.
        if data_type then
            data_json = data_type.label
            content = hex2string(string.gsub(data_json, ":", ""))
            not_uri = true
            content = string.gsub(content, "{", "?")
            content = string.gsub(content, "}", "")
            content = string.gsub(content, ":", "=")
            content = string.gsub(content, ",", "&")
            content = string.gsub(content, "\"", "")
            if (string.find(content, "id_token") == 0) then end
        end
    end
    -- Si no tiene Location en la petición HTTP es que será una petición GET
    -- Guardamos el contenido de la URI
elseif not http_location then
    content = http_uri.value
else return end
-- Comprobamos el request method que se está utilizando
if http_request_method then
        -- Ahora verificamos que sea una petición POST para guardar el contenido
        -- Si es una petición POST será una un Authentication Response
        if http_request_method.value == "POST" then
            -- Convertimos el application/x-www-form-urlencoded de hexadecimal a String y lo tratamos.
            if tcp_data then
                local dato = "" .. tcp_data.label
                content2 =hex2string(string.gsub(dato, ":", ""))
                content = content .."?" .. content2
            end
        end
    end
-- Si no hay coincidencia y no hay contenido en content se sale
if not content then return end
```

Figura 37: Tratamiento de los datos en OIDC

Por otro lado en esta imagen vemos una función que realiza la categorización final de la petición:

```
-- Función que devuelve el tipo de petición que es
-- Como entrada hay que introducirle "s" y "c" ya que dependiendo del tipo de petición HTTP que sea.
function request_type(s,c, m)
    if s == "authorize" then
        -- Comprobamos si la palabra clave "s" es igual a "authorize"
        return petition.AUTHENTICATION_REQUEST, c_auth_request["name"]

    elseif (s == "token" or s == "access_token") then
        -- Comprobamos si la raíz URI es "token" o "access_token"
        return petition.TOKEN_REQUEST, c_token_request["name"]
    elseif string.find(c, "code=") or (string.find(c,"access_token=") and string.find(c, "id_token") and m == 302)then
        -- Comprobamos si hemos añadido a "s" un código que se identifica por empezar con "code="
        -- comprobamos que no sea un Token Request de Authentication Code Flow ya que tienen una estructura similar
        return petition.AUTHENTICATION_RESPONSE, c_auth_response["name"]

    elseif string.find(c,"id_token") and m == 200 then
        -- comprobamos si "s" es igual a "id_token" y si el código de petición es 200
        return petition.TOKEN_RESPONSE, c_token_response["name"]
    elseif string.find(c,"error=") then
        -- Comprobamos si no es una petición de error
        return petition.ERROR, "ERROR"
    end
    -- Si no coincide entonces es otro tipo de petición
    return petition.OTHER, "OTHER"
end
```

Figura 38: Función de catalogación de petición en OIDC

Explicamos ya en el apartado **2.1.1. OpenID Connect** pero se va a definir mejor la función que describimos en la imagen.
La función guarda en el carácter *s* el contenido de la raíz URI, si lo hubiese, y en el carácter *c* el contenido de lo más relevante de cada una de las peticiones.
En el caso de la Petición de Autenticación y la Petición de Token buscamos que coincidan el carácter *s* con los campos que vienen indicados en las especificaciones de OpenID Connect, añadiendo una nueva raíz URI, *access_token* ya que el cliente que estamos utilizando realiza las peticiones con esta extensión.
Por otro lado, para la Respuesta de Autenticación y la Respuesta de Token buscamos ya en el contenido almacenado en el caracte r*c* los valores de *code=* y *id_token* respectivamente.

4. Arquitectura de despliegue

En este apartado trataremos el despliegue de la infraestructura utilizada para las pruebas y la depuración del plugin, documentados en la guía de administración de OpenAM[[17], además de resumir las especificaciones técnicas que se requieren para el despliegue en sí.

4.1. Requisitos funcionales

En esta sección se va a hablar sobre los requisitos funcionales de OpenAM, tanto de software como de hardware.

4.1.1. Requisitos de Hardware

Los requisitos funcionales de hardware para el funcionamiento de la infraestructura, son los siguientes:

- Para la instalación inicial, unos pocos cientos de MB son suficientes, sin incluir los archivos descargados.
- El tamaño de archivo de OpenAM .war varía de una versión a otra, pero en nuestro caso el espacio en disco requerido es de aproximadamente 300 MB.
- Los agentes de políticas se implementan como bibliotecas o aplicaciones web y, por lo tanto, tienden a tener poco tamaño en disco sin superar unos pocos MBs.
- Los servicios básicos de OpenAM requieren un tamaño mínimo de RAM de 1 GB y, cuando se ejecuta en JDK 7, un tamaño de generación permanente mínimo de 256 MB.

4.1.2. Requisitos de Software

Los requisitos funcionales de software para el funcionamiento de la infraestructura son los siguientes:

- OpenAM es compatible con las siguientes versiones del sistema operativo:
  - CentOS 6, 7
  - Microsoft Windows Server 2008, 2008 R2, 2012, 2012 R2
  - Oracle Linux 6, 7
  - Oracle Solaris x64 10, 11
  - Oracle Solaris SPARC 10, 11
  - Red Hat Enterprise Linux 6, 7
  - SuSE Linux 11
  - Ubuntu Linux 12.04 LTS, 14.04 LTS
- OpenAM es combatible con los siguientes servidores web HTTP:

- Apache Tomcat 6, 7, 8
- IBM WebSphere Application Server 8, 8.5
- JBoss Enterprise Application Platform 6
- JBoss Application Server 7
- Oracle WebLogic Server 11g, 12c

- El software de servidor y agente de políticas de OpenAM se basa en Java y para ejecutar necesita un Servidor web, por lo que requiere un Java Development Kit.

4.2. Cliente OpenAM

El cliente OpenAM, tal como explicamos en el apartado de tecnologías, es un servidor web que se puede conseguir de la página web de OpenAM y que ya está configurado para realizar peticiones al servidor, introduciéndole los cambios que necesita el cliente para establecer la conexión.
Ofrece dos modos de autenticación vía web, una mediante el Flujo Código y otra mediante el Flujo Implícito, gracias a lo cual se ha podido implementar y mejorar nuestro plugin.
Tiene un funcionamiento bastante intuitivo y fácil de comprender, además de que ayuda a la hora de entender las trazas enviadas y muestra los principales parámetros para entender mejor el protocolo OpenID Connect sin tener un plugin que analice las trazas.
La configuración necesaria para que se puedan comunicar cliente y servidor dependería del tipo de flujo que utilicemos y sería la siguiente:

4.2.1. Flujo Código:

Primero habría que configurar el agente de OAuth 2.0, tal como explicaremos en el apartado de configuración del servidor OpenAM, **4.3. Servidor OpenAM**, para que use *client_id*, *client_secret* y *redirect_uri*. También habría que añadir como scopes los valores de "openid", que viene por defecto en el agente, y el de "profile". Esto es debido a que la configuración que envía el cliente es la siguiente:

OpenAM URI: /openam
client_id: myClientID
client_secret: password
realm: /
redirect_uri: http://*nuestrohost:puerto*/openid/cb-basic.html

Es necesario también elegir la opción de que cifre el *ID Token* con el algoritmo HS256, para que el cliente pueda descifrar el contenido.
Una vez configurado el servidor con los anteriores parámetros, empezamos la autenticación y se intercambiarán los mensajes entre cliente y servidor, todos ellos explicados en el apartado de *Trazas entre cliente y servidor*, y si la comunicación ha ido bien aparecerá al final un resumen de los parámetro intercambiados tal y

como muestra la **Figura 39**:

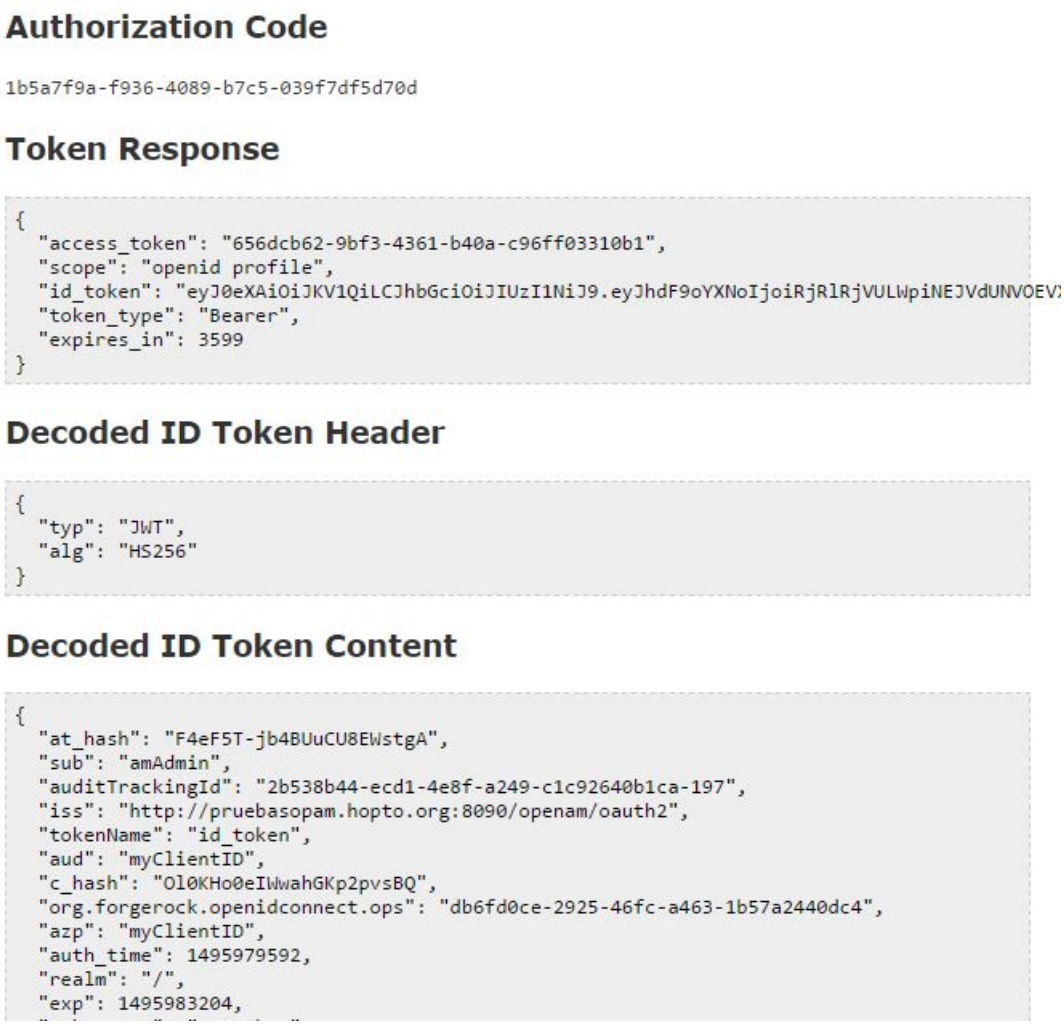

Figura 39: Muestra del mensaje final del cliente

Como se puede observar aparecen los principales campos de OpenID e incluso decodifica la información del *ID Token* para exponer los *claims* intercambiados.

4.2.2. Flujo Implícito:

En este flujo, al igual que con el anterior, es necesario configurar el agente de OAuth 2.0 con la información de "client_id" y "redirect_uri", además de configurar los valores del scope para que devuelva "openid" y "profile". La configuración que necesita el cliente para intercambiar los mensajes con el servidor serían los siguientes:

OpenAM URI: /openam
client_id: myClientID
realm: /
redirect_uri: http://*nuestrohost:puerto*/openid/cb-implicit.html

Al igual que en el caso del Flujo Código, hay que elegir que el *ID Token* sea cifrado con el algoritmo HS256. Una vez se haya autenticado de la forma adecuada y si se ha configurado correctamente, aparecerá un resumen como el mostrado en el Flujo Código.

4.3. Servidor OpenAM

Una vez conocidos las exigencias del Cliente OpenAM, tal como se ha mencionado en el apartado anterior, se necesita configurar el servidor OpenAM correctamente para que se pueda realizar la conexión entre ambas partes.
Aquí no se hace referencia a la instalación del servidor ya que todo se detalla en el anexo *1. Instalación del Servidor OpenAM* por lo que se pasa directamente a exponer los pasos a seguir para la configuración.

4.3.1. Seleccionar el Realm

Una vez se autentica como administrador, lo primero sería acceder al realm raíz que pide el cliente OpenAM en ambos flujos de autenticación, *realm: /*:

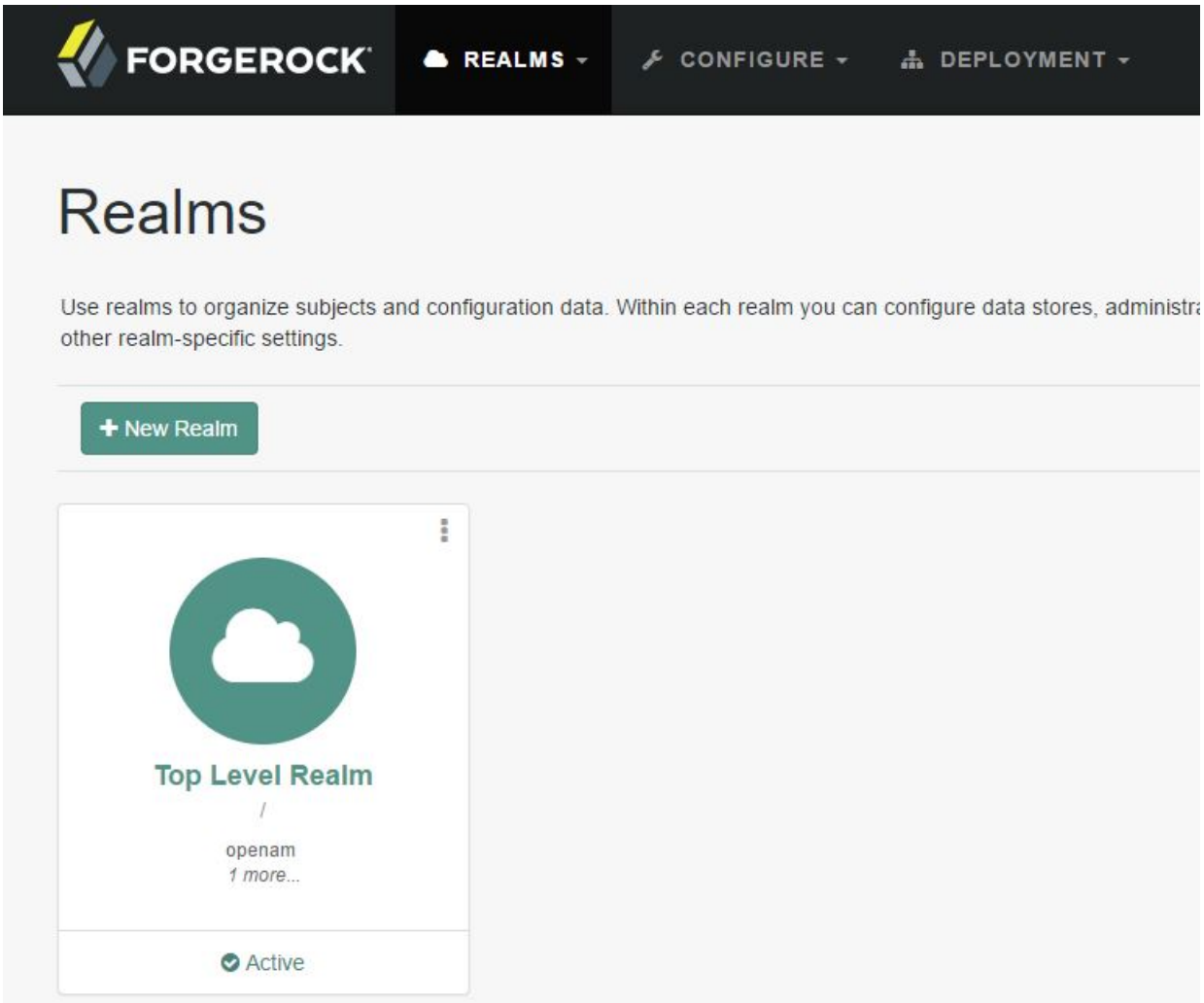

Figura 40: Seleccionar el realm del Servidor

4.3.2. Usuario a autenticar

Para personalizar los usuarios y para realizar las pruebas, se configura uno aparte de los usuarios creados por el servidor por defecto. Para ello, se pulsa en la opción *subject* del menú de la izquierda de OpenAM. Una vez pulsado, se despliega un menú con los usuarios creados para la autenticación en el servidor:

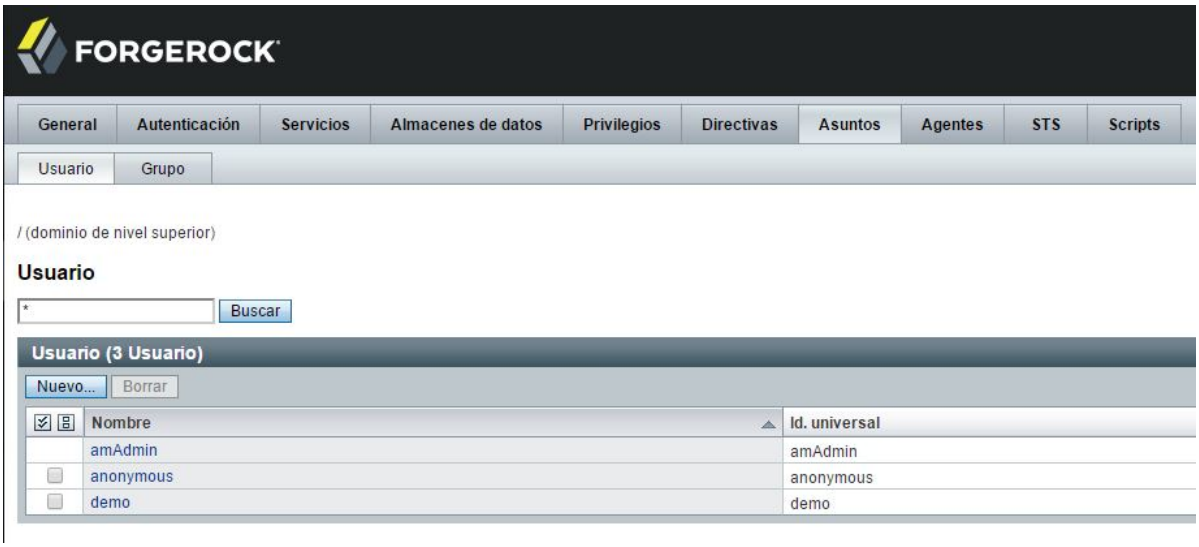

Figura 41: Usuarios de OpenAM por defecto

Como se puede observar en la **Figura 40** existe la opción de *nuevo* para poder configurar usuarios adicionales:

Figura 42: Creación de un usuario nuevo

Una vez creado un usuario nuevo, ya podemos acceder a su configuración y poder completarla. Esa información será la que entregue el servidor OpenAM al cliente si así lo especifica entre sus *claims*:

Figura 43: Parte de la configuración del cliente

4.3.3. Configuración del Agente

Una vez creado el nuevo usuario, es hora de configurar el agente. Esta configuración se realiza desde una opción de menú que se llama *Agentes* y se encuentra en la parte superior de OpenAM. Una vez seleccionamos la opción se desplegará un submenú y se tendrá que escoger la opción de *Oauth 2.0/OpenID Connect Client*:

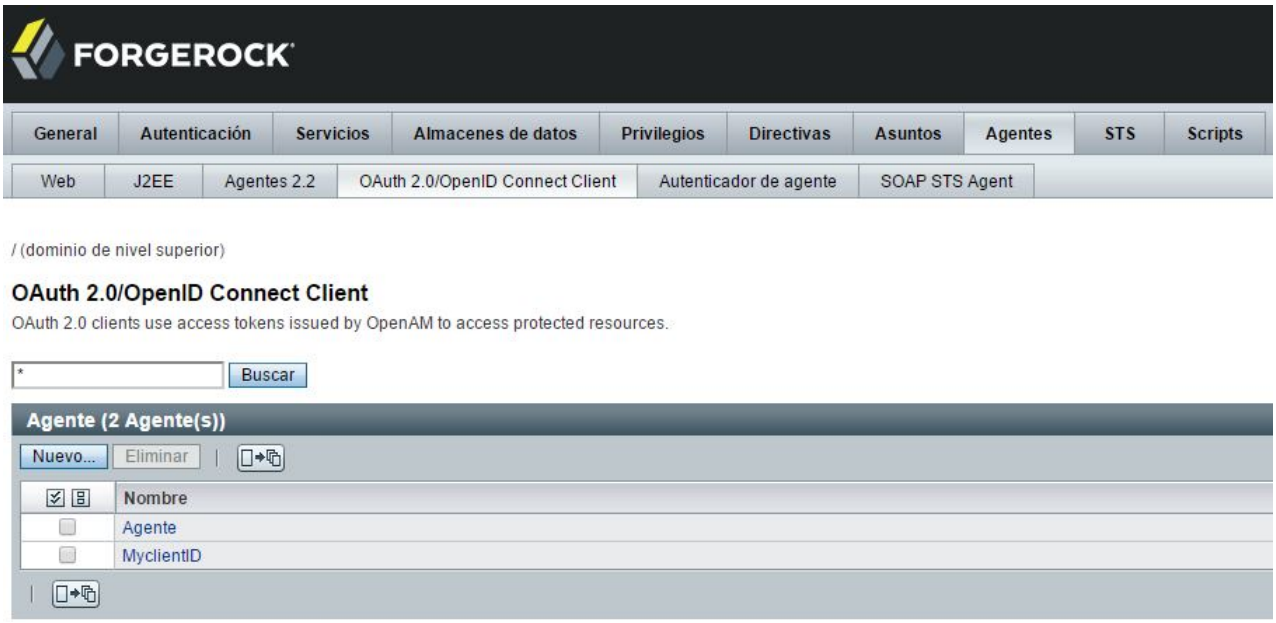

Figura 44: Submenú de Agentes

Ahora se necesita crear un nuevo Agente tal como lo especifica el cliente OpenAM, es decir, que tenga las siguientes características:

- Nombre: MyClientID
  Nota aclaratoria: el servidor no diferencia entre mayúsculas y minúsculas en el nombre, por lo que para él será el mismo agente uno con nombre MyClientID que otro con nombre myclientid.
- Contraseña: password

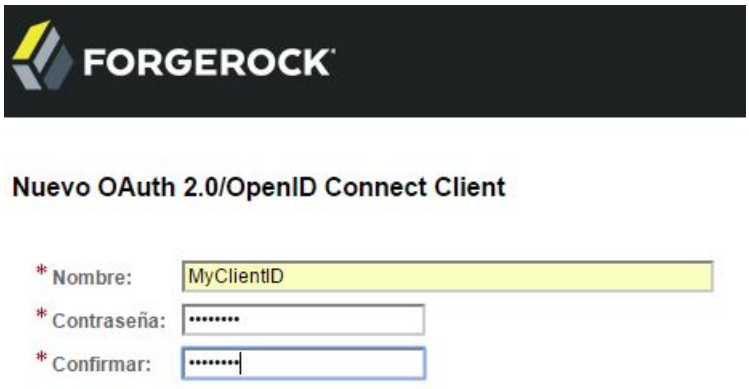

Figura 45: Creación de un Agente

Una vez creado se va a realizar la configuración según las especificaciones del cliente para ambos flujos, el Flujo Código y el Flujo Implícito:

- Configuraremos primero las direcciones de redirección:

Figura 46: Configuración de las URIs de redirección

- Añadiremos al scope los *claims* necesarios para la autenticación:

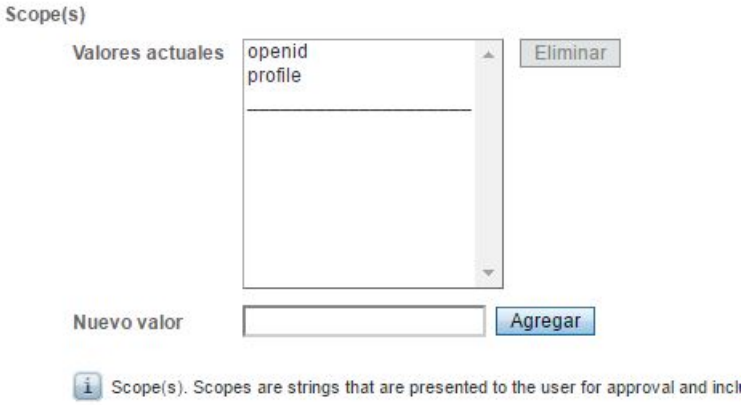

Figura 47: Configuración del Scope

- Por último confiugraremos el algoritmo de cifrado de la firma del *ID Token*, que según especificaba el cliente tenía que ser HS256:

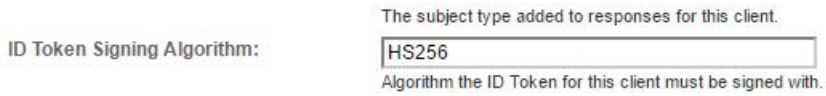

Figura 48: Configuración del algoritmo de cifrado

Siguiendo estos pasos configuraremos adecuadamente el servidor para que se pueda comunicar con el cliente.

Ahora que conocemos más en detalle el funcionamiento del servidor y el cliente, el funcionamiento del protocolo OpenID Client y la diferencia que existe entre los distintos flujos, se van a explicar las trazas intercambiadas entre ambas partes para fortalecer los conocimientos con un ejemplo gráfico.

4.4. Trazas entre Cliente y Servidor

1. Una vez escogemos el flujo con el que se inicia la autenticación del cliente, el Cliente o RP enviará un redireccionamiento al usuario final para forzar el despliegue del menú de autenticación:

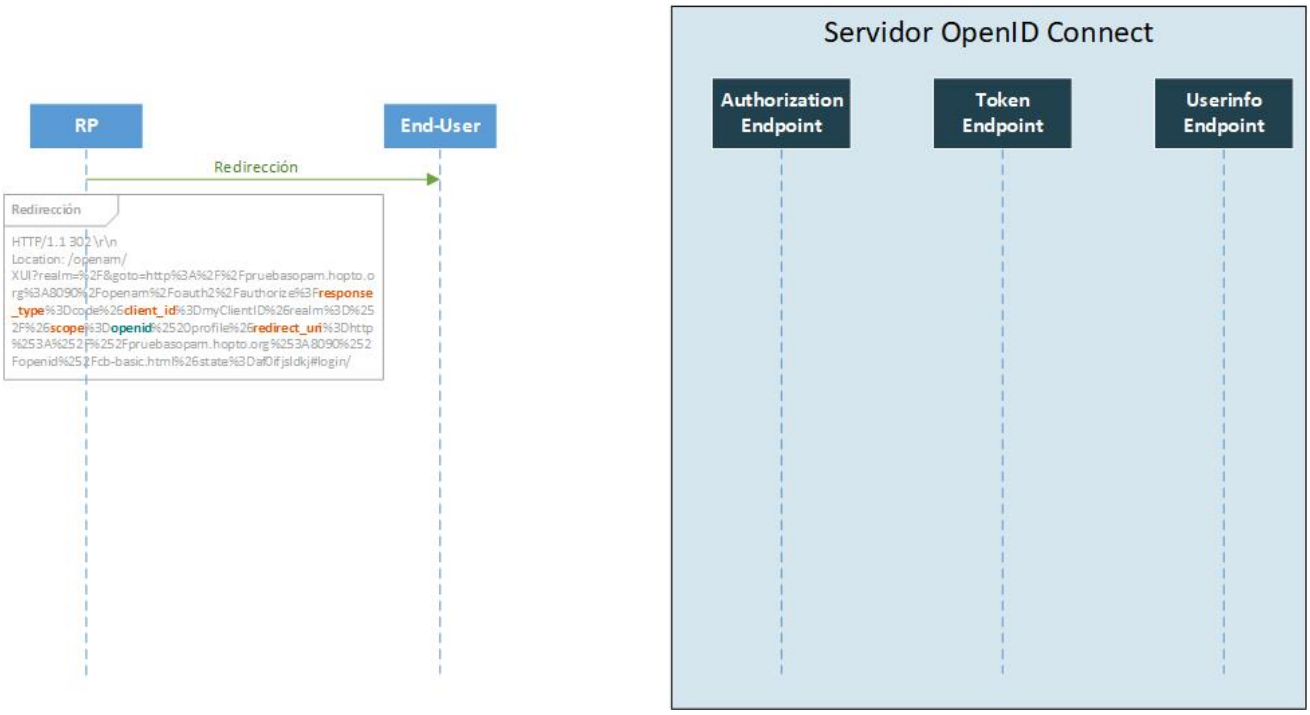

Figura 49: Redirecciónamiento por parte del Cliente

Se puede apreciar en la **Figura 49** cómo se envían los principales parámetros que comentábamos en el apartado *2.1.1 OpenID Connect*:

- response_type = code
  Indica con este campo que el flujo escogido es el Flujo Código.
- client_id = myClientID
  Tal como se explica en la configuración del cliente, este parámetro es igual a myClientID.
- scope = openid profile
  En este caso y como pide el cliente configuramos los dos scope's, openid y profile.
- redirect_uri
  Dirección del cliente a donde tiene que reenviar la respuesta.

2. Una vez se ha forzado que se haga la redirección, el usuario envía una petición de autenticación al servidor OpenAM:

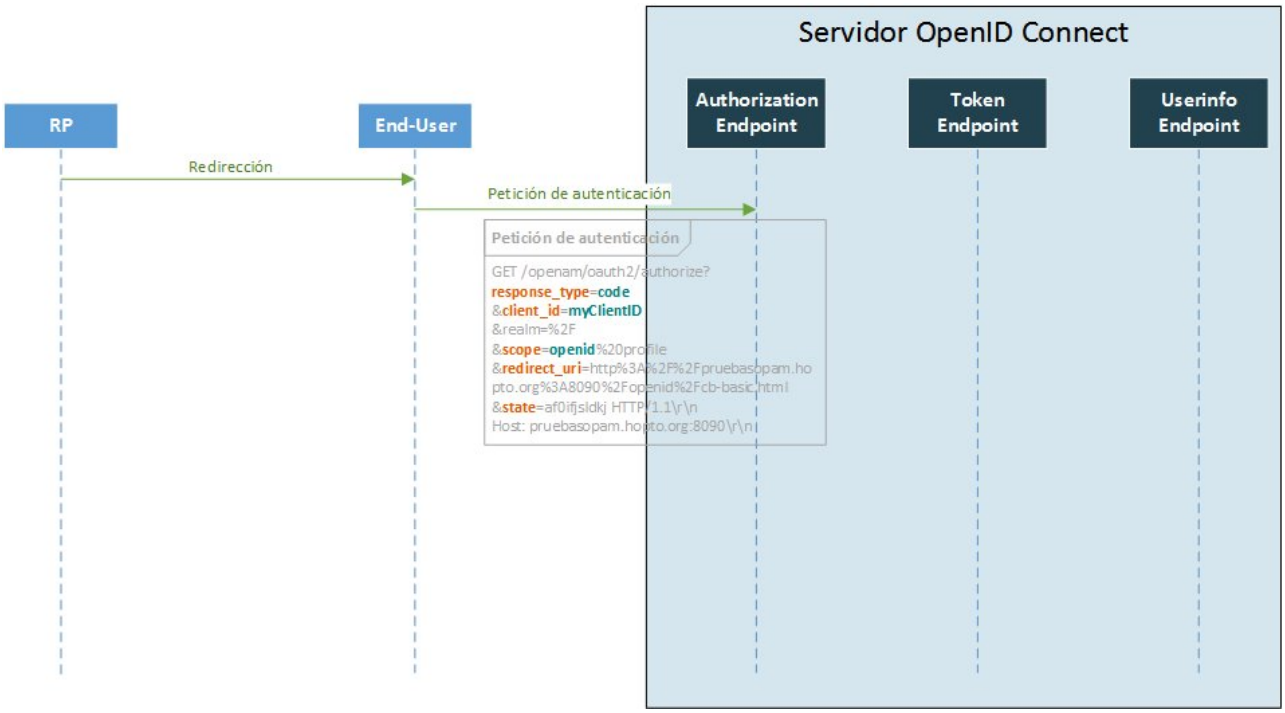

Figura 50: Envío de la Petición de Autenticación

En este caso se puede apreciar, en la **Figura 50**, que manda los *claims* obligatorios que se explicaron en el apartado de OpenID Connect, que son exactamente los mismos que en la petición de redirección.

3. El siguiente paso es que el usuario final se autentique en el servidor OpenAM. Por ello, si las peticiones y la configuración han sido las correctas, aparecerá el siguiente menú de autenticación, como se muestra en la **Figura 51**:

Figura 51: Menú de autenticación en el servidor

4. Una vez autenticado con éxito se pide al usuario final que de su consentimiento para el envío de datos al cliente, como se observa en la **Figura 52**:

Figura 52: Consentimiento del usuario final

5. Una haya confirmado el usuario final que consiente el envío de su información, el servidor OpenAM enviará una respuesta de autenticación y redirigirá al usuario final a la dirección de redirección que aparecía en la petición de autenticación, en el campo *redirect_uri*:

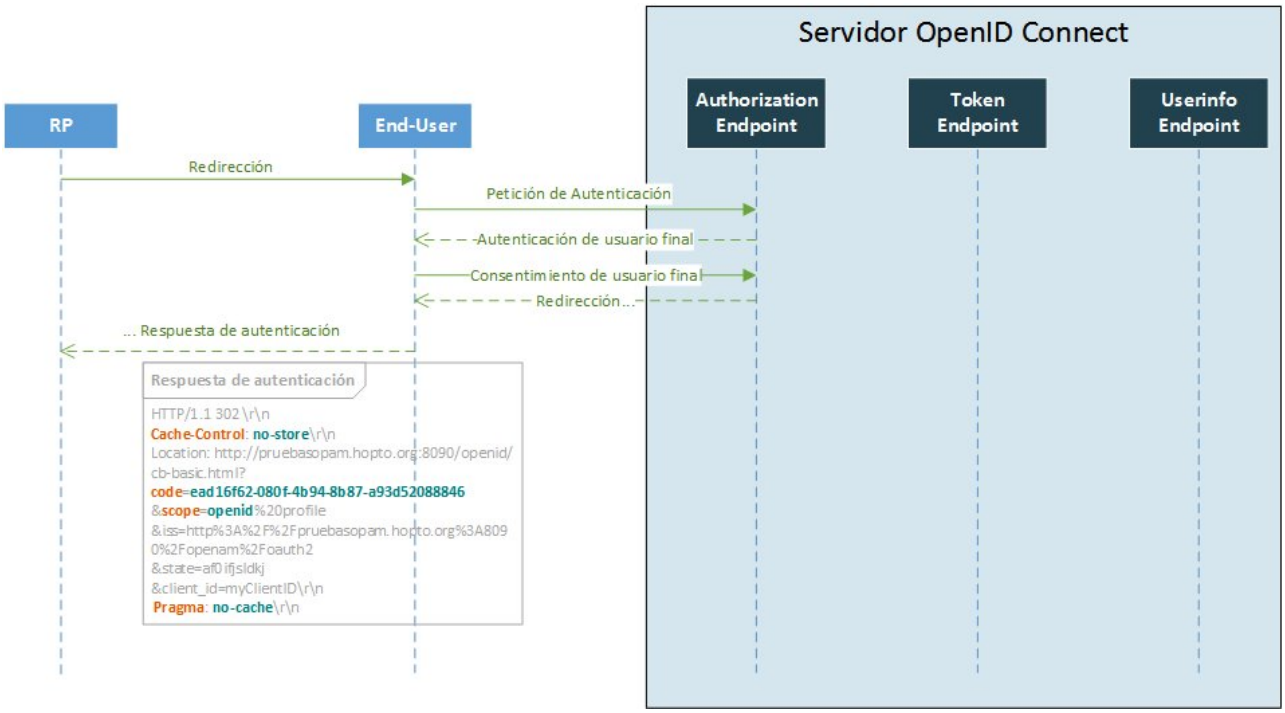

Figura 53: Redireccionamiento y nvío de la Respuesta de Autenticación

Como se puede apreciar en la respuesta de autenticación, ésta contiene los parámetros obligatorios que se explicaron con anterioridad:

- Las cabeceras obligatorias al enviar una petición que contiene alguna petición con un Token:
  Cache-Contro: no-store
  Pragma: no-cache
- Code: devuelve un código de autenticación para posteriormente conseguir el *ID Token* y el Token de acceso.
- Scope = openid profile
- Resto de *claims* que no son obligatorios, pero algunos, como es el caso de state, son recomendados.

6. El siguiente paso consiste en que el cliente envía al *Token Endpoint* una Petición de Autenticación, añadiendo el code que le ha enviado, como la que se muestra en la **Figura 54**n:

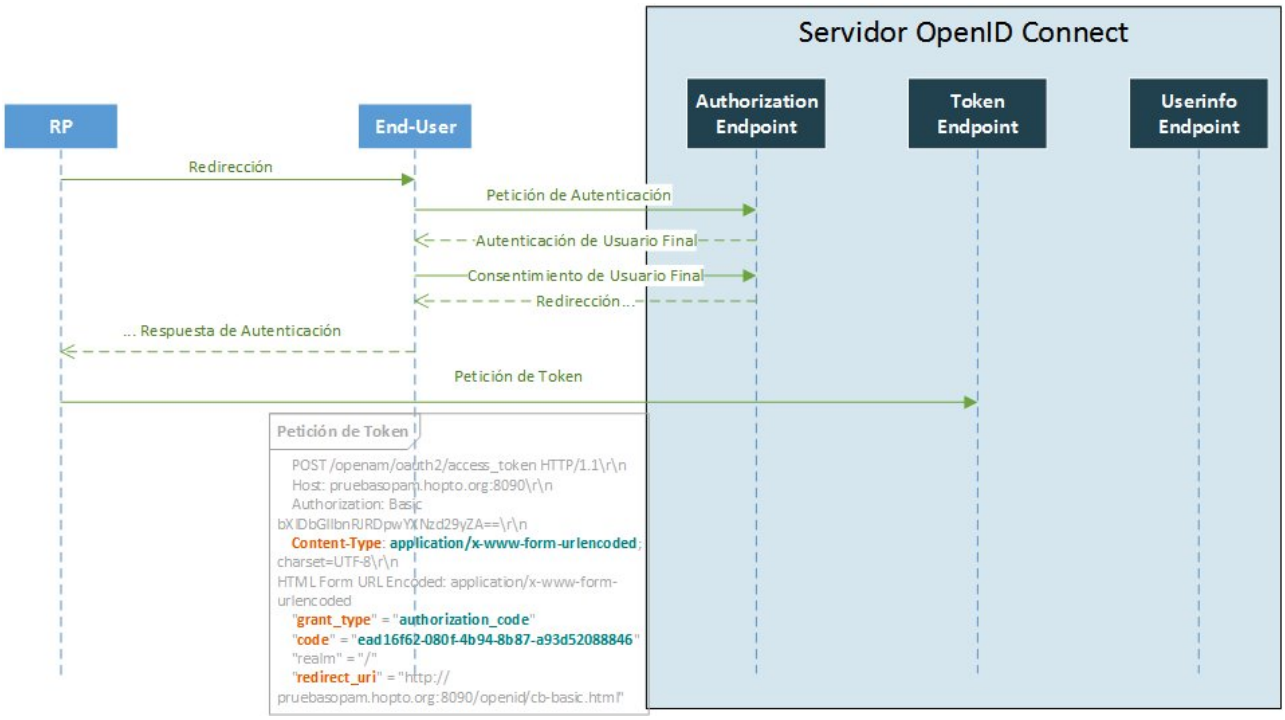

Figura 54: Envío de Petición de Token

Como se puede observar, es una petición POST de tipo application/x-www-form-urlencoded, tal como se especifica en OpenID Connect. Tiene los siguientes campos importantes:

- grant_type = authorization_code
- code: se vuelve a enviar el mismo code que le envió el servidor OpenAM en la Respuesta de Autenticación.
- redirect_uri
  Se vuelve a indicar cuál es la dirección de respuesta a la que hay que enviar la Respuesta de Token.

7. Una vez recibida la petición por parte el servidor OpenAM, le devuelve al cliente una Respuesta de Token. Se puede apreciar que es una respuesta HTTP con el código 200 y el tipo media es de application/json:

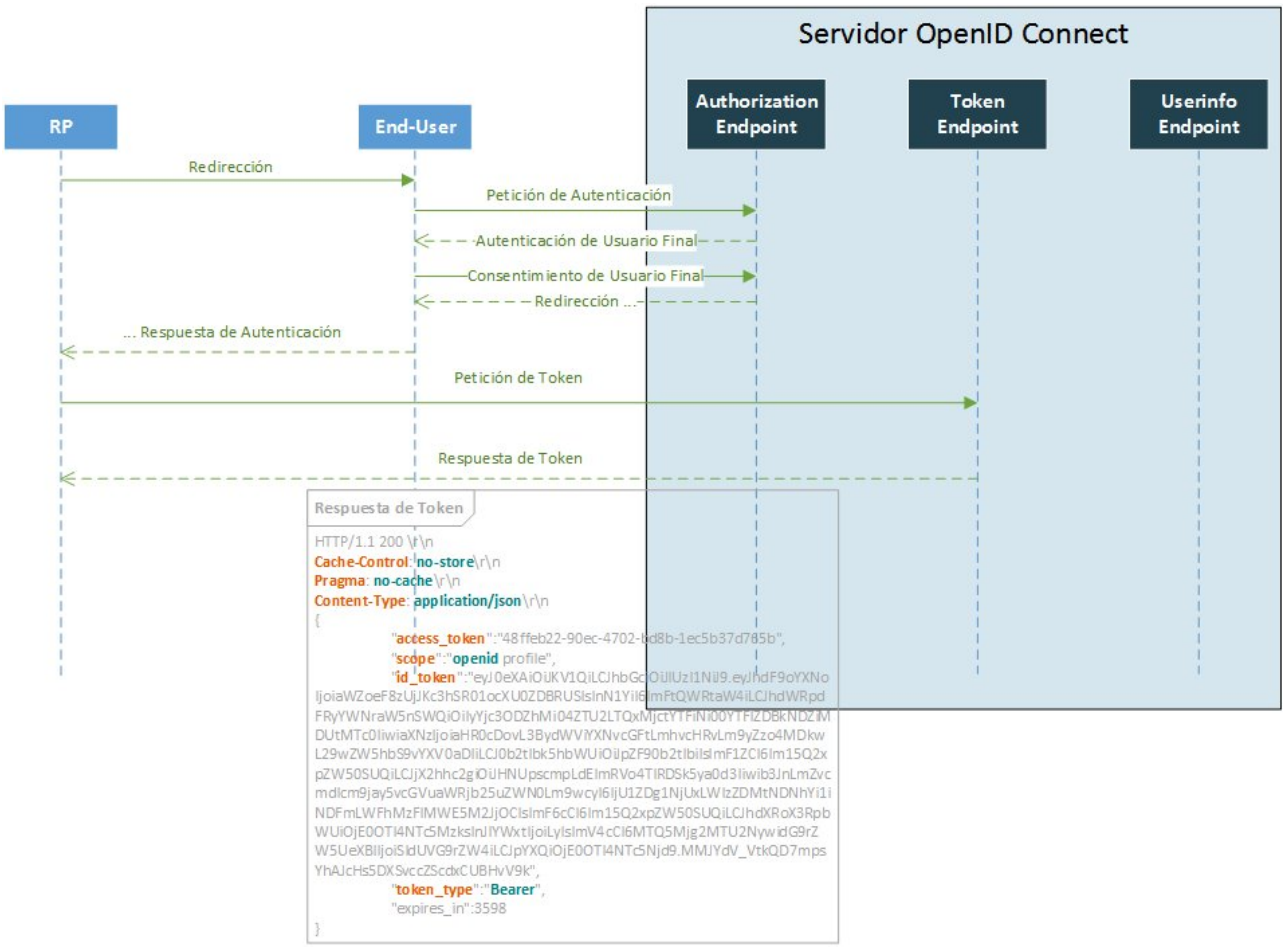

Figura 55: Envío de Respuesta de Token

En esta respuesta le envía el Token de Acceso y el *ID Token* codificado, además de los siguientes campos:

- Las cabeceras obligatorias al enviar una petición que contiene alguna petición con un Token:
  Cache-Contro: no-store
  Pragma: no-cache
- token_type : Bearer
- expires_in: tiempo límite que dan desde que se ha generado el Token hasta que lo puedas utilizar.

Una vez se tenga el *ID Token* únicamente queda descodificarlo para conocer los campos importantes:

```
{
  "typ": "JWT",
  "alg": "HS256"
}

{
  "at_hash": "ifhx_3R2JsxRGMhqu4d0QQ",
  "sub": "amAdmin",
  "auditTrackingId": "2b7786a2-8e56-4127-a1b6-4a1ed0d46b05-174",
  "iss": "http://pruebasopam.hopto.org:8090/openam/oauth2",
  "tokenName": "id_token",
  "aud": "myClientID",
  "c_hash": "G5JlrjKtIfEZ8NTCJNrkGw",
  "org.forgerock.openidconnect.ops": "55d85651-b3d3-43ab-b41f-aa31e1a93bc8",
  "azp": "myClientID",
  "auth_time": 1492857939,
  "realm": "/",
  "exp": 1492861567,
  "tokenType": "JWTToken",
  "iat": 1492857967
}

MMJYdV_VtkQD7mpsYhAJcHs5DXSvccZScdxCUBHvV9k
```


Figura 56: *ID Token* descodificado

Estas son las trazas importantes que puede ofrecer el cliente OpenAM interaccionando con el servidor OpenAM, que dan una visión más clara sobre el protocolo OpenID Client.

5. Pruebas

En este apartado se comentan las pruebas que se han realizado para la mejora de nuestro plugin de OpenID Connect haciendo una diferenciación de escenarios, primero uno local y después uno distribuido.
Cabe destacar el hecho de que se han realizado las mismas pruebas para un escenario que para el otro, y que únicamente se testea el comportamiento de nuestro disector ante las distintas trazas, sin entrar en temas sobre el correcto funcionamiento del cliente y el servidor.

5.1. Escenario local

En este escenario se recrea el mismo cliente y servidor del cual se ha hablado en la sección **4. Arquitectura del despliegue** y se realizan distintas pruebas para la comprobación del plugin utilizando un mismo equipo para la autenticación, es decir, utilizando el equipo en el cual tenemos desplegado toda la infraestructura.
Se estructuran las pruebas realizadas en este apartado primero detallando las pruebas de infraestructura entre cliente y servidor y después tratamos las pruebas divididas por tipo de peticiones existentes en OpenID Connect que realice nuestro cliente OpenAM y de los flujos que soporta nuestro cliente:

5.1.1. Pruebas de infrestraestructura

En este apartado se tratan las pruebas realizadas a la hora de autenticar en el servidor para comprobar la robustez y los mensajes de error de OpenID Connect:

Prueba	**Éxito**	**Descripción del resultado**
Autenticación en el servidor OpenAM de forma correcta	Si	El resultado mostrado concuerda con los resultados esperados de OpenID Connect, recibiendo los *Tokens* necesarios en el cliente
Autenticación en el servidor OpenAM con credenciales incorrectos	Si	El resultado mostrado concuerda con los resultados esperados de OpenID Connect, recibiendo un error por parte del servidor
Esperar a la hora de terminar el proceso de autenticación en el servidor	Si	El resultado mostrado concuerda con los resultados esperados de OpenID Connect, recibiendo un error debido a que habían expirado los *Tokens*

Tabla 11: Pruebas de infraestructura en local

Como se puede observar, el cliente y servidor actuan respetando las especificaciones de OpenID Connect y recibimos los mensajes de las pruebas realizadas acorde a ello.

5.1.2. Pruebas de Petición de Autenticación

En este apartado ya empezamos a describir las pruebas realizadas sobre el mismo plugin, para comprobar los resultados que muestra en Wireshark al analizar trazas de Peticiones de Autenticación:

Prueba	Éxito	Descripción del resultado
Autenticación en el servidor OpenAM de forma correcta en el Flujo Código	Si	El plugin categoriza correctamente la traza de Petición de Autenticación y el tipo de flujo
Autenticación en el servidor OpenAM de forma correcta en el Flujo Implícito	Si	El plugin categoriza correctamente la traza de Petición de Autenticación y el tipo de flujo
Autenticación en el servidor OpenAM con credenciales incorrectos en el flujo Flujo Código	Si	El resultado mostrado concuerda con los resultados esperados de nuestro plugin, categorizándolo como una petición "ERROR"
Autenticación en el servidor OpenAM con credenciales incorrectos en el Flujo Implícito	Si	El resultado mostrado concuerda con los resultados esperados de nuestro plugin, categorizándolo como una petición "ERROR"
Comprobar si añade correctamente los parámetros relevantes	Si	Añade correctamente los parámetros de este tipo de petición en el flujo Flujo Código
Comprobar si añade correctamente los parámetros relevantes	Si	Añade correctamente los parámetros de este tipo de petición en el Flujo Implícito

Tabla 12: Pruebas de Petición de Autenticación en local

Como se ha podido observar por las pruebas realizadas, se comporta como esperábamos y categoriza correctamente este tipo de peticiones para ambos tipos de flujos tratados.

5.1.3. Pruebas de Respuesta de Autenticación

En este apartado describiremos las pruebas realizadas sobre el plugin, para comprobar los resultados que muestra en Wireshark al analizar trazas de Respuesta de Autenticación:

Prueba	Éxito	Descripción del resultado
Autenticación en el servidor OpenAM de forma correcta en el flujo Flujo Código	Si	El plugin categoriza correctamente la traza de Respuesta de Autenticación y el tipo de flujo
Autenticación en el servidor OpenAM de forma correcta en el Flujo Implícito	Si	El plugin categoriza correctamente la traza de Respuesta de Autenticación y el tipo de flujo
Autenticación en el servidor OpenAM con credenciales incorrectos en el Flujo Código	Si	El resultado mostrado concuerda con los resultados esperados de nuestro plugin, categorizándolo como una petición "ERROR"
Autenticación en el servidor OpenAM con credenciales incorrectos en el Flujo Implícito	Si	El resultado mostrado concuerda con los resultados esperados de nuestro plugin, categorizándolo como una petición "ERROR"
Comprobar si añade correctamente los parámetros	Si	Añade correctamente los parámetros de este tipo de petición en el Flujo Código
Comprobar si añade correctamente los parámetros	Si	Añade correctamente los parámetros de este tipo de petición en el Flujo Implícito
Añade correctament el parámetro *Code* a la petición	Si	Añade correctamente el parámetro de este tipo de petición en el Flujo Código
Comprobar que recoge el parámetro *id_token* en la petición en el Flujo Implícito	Si	Añade correctamente el parámetro de este tipo de petición en el Flujo Implícito
Comprobar que decodifica el parámetro *id_token* en la petición y lo añade al árbol en el Flujo Implícito	Si	Añade correctamente el parámetro de este tipo de petición en el Flujo Implícito

Tabla 13: Pruebas de Respuesta de Autenticación en local

Como se ha podido observar por las pruebas realizadas, se comporta como esperábamos y categoriza correctamente este tipo de peticiones para ambos tipos de flujos tratados.

5.1.4. Pruebas de Petición de Token

En este apartado describiremos las pruebas realizadas sobre el plugin, para comprobar los resultados que muestra en Wireshark al analizar trazas de Petición de Token:

Prueba	Éxito	Descripción del resultado
Autenticación en el servidor OpenAM de forma correcta en el Flujo Código	Si	El plugin categoriza correctamente la traza de Respuesta de Autenticación y el tipo de flujo
Autenticación en el servidor OpenAM con credenciales incorrectos en el Flujo Código	Si	El resultado mostrado concuerda con los resultados esperados de nuestro plugin, categorizándolo como una petición "ERROR"
Comprobar si añade correctamente los parámetros	Si	Añade correctamente los parámetros de este tipo de petición en el Flujo Código
Comprobar si se ha añadido el contenido de application/x-www-form-urlencoded	No	Depende de como se reciba en la petición el contenido de x-www-form-rulencoded el plugin lo recoge o no

Tabla 14: Pruebas de Petición de Token en local

Como se ha podido observar por las pruebas realizadas, se comporta como esperábamos y categoriza correctamente este tipo de peticiones para ambos tipos de flujos tratados, menos a la hora de analizar los parámetros contenidos en el application/x-www-form-urlencoded ya que a veces se encuentra contenido en el disector *TCP.segmented_data* y otras veces no.
Igualmente el plugin puede realizar su categorización adecuadamente sin tener en cuenta estos datos y en la interfaz de Wireshark se pueden observar todos los parámetros en el disector con el mismo nombre, tal como se muestra en la **Figura 57**:

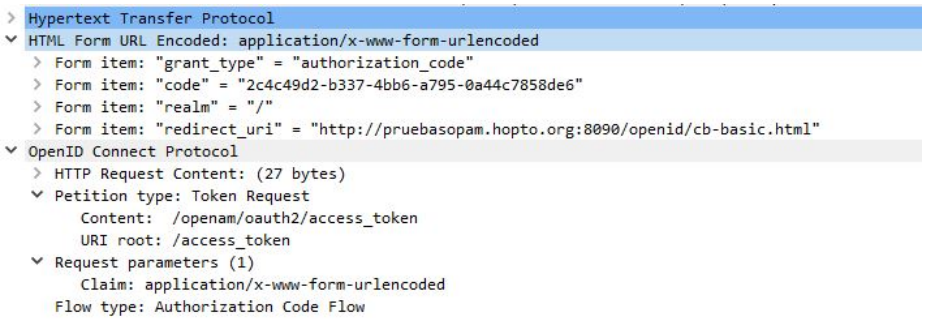

Figura 57: Infterfaz de Wireshark en Peticiones de Token

5.1.5. Pruebas de Respuesta de Token

En este apartado describiremos las pruebas realizadas sobre el plugin, para comprobar los resultados que muestra en Wireshark al analizar trazas de Respuesta de Autenticación:

Prueba	Éxito	Descripción del resultado
Autenticación en el servidor OpenAM de forma correcta en el Flujo Código	Si	El plugin categoriza correctamente la traza de Respuesta de Autenticación y el tipo de flujo
Autenticación en el servidor OpenAM con credenciales incorrectos en el Flujo Código	Si	El resultado mostrado concuerda con los resultados esperados de nuestro plugin, categorizándolo como una petición "ERROR"
Comprobar si añade correctamente los parámetros en el Flujo Código	Si	Añade correctamente los parámetros de este tipo de petición en el Flujo Código
Comprobar que recoge el parámetro *id_token* en la petición en el Flujo Código	Si	Añade correctamente el parámetro de este tipo de petición en el Flujo Código
Comprobar que decodifica el parámetro *id_token* en la petición y lo añade al árbol en el Flujo Código	Si	Añade correctamente el parámetro de este tipo de petición en el Flujo Código

Tabla 15: Pruebas de Respuesta de Token en local

Como se ha podido observar por las pruebas realizadas, se comporta como esperábamos y categoriza correctamente este tipo de peticiones para ambos tipos de flujos tratados.

5.2. Escenario Distribuido

En este escenario se recrea el mismo cliente y servidor del cual se ha hablado en la sección **4. Arquitectura del despliegue** y se han realizado distintas pruebas para la comprobación del plugin utilizando dos equipos para la autenticación, un equipo en el cual se realiza la conexión al cliente y otro en el que tenemos desplegado nuestro servidor, donde además tendremos nuestro Wireshark corriendo.
A diferencia del apartado anterior, se van a realizar las pruebas únicamente dependiendo del tipo de peticiones existentes en OpenID Connect que realice nuestro cliente OpenAM y no se va a hacer mención de las pruebas de infraestructura dado que son las mismas.

5.2.1. Pruebas de Petición de Autenticación

En este apartado ya empezamos a describir las pruebas realizadas sobre el mismo plugin, para comprobar los resultados que muestra en Wireshark al analizar trazas de Peticiones de Autenticación:

Prueba	Éxito	Descripción del resultado
Autenticación en el servidor OpenAM de forma correcta en el Flujo Código	Si	El plugin categoriza correctamente la traza de Petición de Autenticación y el tipo de flujo
Autenticación en el servidor OpenAM de forma correcta en el Flujo Implícito	Si	El plugin categoriza correctamente la traza de Petición de Autenticación y el tipo de flujo
Autenticación en el servidor OpenAM con credenciales incorrectos en el flujo Flujo Código	Si	El resultado mostrado concuerda con los resultados esperados de nuestro plugin, categorizándolo como una petición "ERROR"
Autenticación en el servidor OpenAM con credenciales incorrectos en el Flujo Implícito	Si	El resultado mostrado concuerda con los resultados esperados de nuestro plugin, categorizándolo como una petición "ERROR"
Comprobar si añade correctamente los parámetros relevantes	Si	Añade correctamente los parámetros de este tipo de petición en el flujo Flujo Código
Comprobar si añade correctamente los parámetros relevantes	Si	Añade correctamente los parámetros de este tipo de petición en el Flujo Implícito

Tabla 16: Pruebas de Petición de Autenticación en entorno distribuido

Como se ha podido observar por las pruebas realizadas, se comporta como esperábamos y categoriza correctamente este tipo de peticiones para ambos tipos de flujos tratados.

5.2.2. Pruebas de Respuesta de Autenticación

En este apartado describiremos las pruebas realizadas sobre el plugin, para comprobar los resultados que muestra en Wireshark al analizar trazas de Respuesta de Autenticación:

Prueba	Éxito	Descripción del resultado
Autenticación en el servidor OpenAM de forma correcta en el flujo Flujo Código	Si	El plugin categoriza correctamente la traza de Respuesta de Autenticación y el tipo de flujo
Autenticación en el servidor OpenAM de forma correcta en el Flujo Implícito	Si	El plugin categoriza correctamente la traza de Respuesta de Autenticación y el tipo de flujo
Autenticación en el servidor OpenAM con credenciales incorrectos en el Flujo Código	Si	El resultado mostrado concuerda con los resultados esperados de nuestro plugin, categorizándolo como una petición "ERROR"
Autenticación en el servidor OpenAM con credenciales incorrectos en el Flujo Implícito	Si	El resultado mostrado concuerda con los resultados esperados de nuestro plugin, categorizándolo como una petición "ERROR"
Comprobar si añade correctamente los parámetros	Si	Añade correctamente los parámetros de este tipo de petición en el Flujo Código
Comprobar si añade correctamente los parámetros	Si	Añade correctamente los parámetros de este tipo de petición en el Flujo Implícito
Añade correctament el parámetro *Code* a la petición	Si	Añade correctamente el parámetro de este tipo de petición en el Flujo Código
Comprobar que recoge el parámetro *id_token* en la petición en el Flujo Implícito	Si	Añade correctamente el parámetro de este tipo de petición en el Flujo Implícito
Comprobar que decodifica el parámetro *id_token* en la petición y lo añade al árbol en el Flujo Implícito	Si	Añade correctamente el parámetro de este tipo de petición en el Flujo Implícito

Tabla 17: Pruebas de Respuesta de Autenticación en entorno distribuido

Como se ha podido observar por las pruebas realizadas, se comporta como esperábamos y categoriza correctamente este tipo de peticiones para ambos tipos de flujos tratados.

5.2.3. Pruebas de Petición de Token

En este apartado describiremos las pruebas realizadas sobre el plugin, para comprobar los resultados que muestra en Wireshark al analizar trazas de Petición de Token:

Prueba	Éxito	Descripción del resultado
Autenticación en el servidor OpenAM de forma correcta en el Flujo Código	Si	El plugin categoriza correctamente la traza de Respuesta de Autenticación y el tipo de flujo
Autenticación en el servidor OpenAM con credenciales incorrectos en el Flujo Código	Si	El resultado mostrado concuerda con los resultados esperados de nuestro plugin, categorizándolo como una petición "ERROR"
Comprobar si añade correctamente los parámetros	Si	Añade correctamente los parámetros de este tipo de petición en el Flujo Código
Comprobar si se ha añadido el contenido de application/x-www-form-urlencoded	No	Depende de como se reciba en la petición el contenido de x-www-form-rulencoded el plugin lo recoge o no

Tabla 18: Pruebas de Petición de Token en entorno distribuido

Como se ha podido observar por las pruebas realizadas, se comporta como esperábamos y categoriza correctamente este tipo de peticiones para ambos tipos de flujos tratados, menos a la hora de analizar los parámetros contenidos en el application/x-www-form-urlencoded como pasaba con la pruebas en local.

5.2.4. Pruebas de Respuesta de Token

En este apartado describiremos las pruebas realizadas sobre el plugin, para comprobar los resultados que muestra en Wireshark al analizar trazas de Respuesta de Autenticación:

Prueba	Éxito	Descripción del resultado
Autenticación en el servidor OpenAM de forma correcta en el Flujo Código	Si	El plugin categoriza correctamente la traza de Respuesta de Autenticación y el tipo de flujo
Autenticación en el servidor OpenAM con credenciales incorrectos en el Flujo Código	Si	El resultado mostrado concuerda con los resultados esperados de nuestro plugin, categorizándolo como una petición "ERROR"
Comprobar si añade correctamente los parámetros en el Flujo Código	Si	Añade correctamente los parámetros de este tipo de petición en el Flujo Código
Comprobar que recoge el parámetro *id_token* en la petición en el Flujo Código	Si	Añade correctamente el parámetro de este tipo de petición en el Flujo Código
Continuación de la tabla		

Tabla 19 – Continuación de las pruebas de Respuesta de Token

Prueba	Éxito	Descripción del resultado
Comprobar que decodifica el parámetro *id_token* en la petición y lo añade al árbol en el Flujo Código	Si	Añade correctamente el parámetro de este tipo de petición en el Flujo Código

Tabla 19: Pruebas de Respuesta de Token en entorno distribuido

Como se ha podido observar por las pruebas realizadas, se comporta como esperábamos y categoriza correctamente este tipo de peticiones para ambos tipos de flujos tratados.

Con esto concluimos con que las pruebas han sido exitosas y el plugin se comporta tal como esperábamos de forma genérica; categoriza adecuadamente por tipo de petición en todas las pruebas realizadas y por tipo de flujo, aunque es cierto que algún tipo de petición las evidencias mostradas no han sido totalmente las deseadas, creemos que el plugin cumple totalmente con su función.

6. Planificación y presupuesto

En este apartado se exponen las etapas de la planificación que se han llevado a cabo para el desarrollo del proyecto, además de realizar un presupuesto que recoge todos los costes que se han tenido en cuenta para su realización.

6.1. Planificación

El objetivo de la planificiación es establecer fechas de entrega del proyecto en sí y la de definición de las tareas. Por ello, como punto inicial se tendá en cuenta el primer contacto mantenido para la realización del proyecto, que fue el 2 de diciembre de 2016, donde se decidió que sería interesante la realización de un analizador de trazas de OpenID Connect y que ha durado hasta el día 20 de julio del 2017.
Es cierto que la duración del proyecto fue larga, pero las horas activas a las que se les ha dedicado es inferior a esa cantidad, por lo que se podría decir que el proyecto tuvo una duración total de 12 meses reales además de que no se dedicó el 100 % del tiempo en la realización del mismo, lo que se tendrá en cuenta a la hora del cálculo del presupuesto.

6.1.1. Etapas del proyecto

Se puede explicar todo el proceso del proyecto desglosándolo en cuatro fases que reflejan la aproximación desde el comienzo:

- Fase de análisis de requisitos
  En esta fase se realizó un análisis de la infraestructura a utilizar, los requisitos que se querían reunir para el análisis de las trazas, el programa que mejor reflejaría la trazabilidad y del lenguaje de desarrollo que utilizaríamos para su implementación.

- Fase de diseño y arquitectura
  Una vez se realiza el análisis del proyecto se lleva todas las ideas a tierra y se escoge una arquitectura que nos ayude a recrear el entorno ideal para la realización del plugin, además de realizar un estudio para diseñarlo de la forma más didáctica e intuitiva.

- Fase de desarrollo
  En esta fase realizamos el desarrollo del plugin de OpenID Connect y además engloba las pruebas realizadas para su correcto funcionamiento. Además se han realizado cambios respecto a lo acordado en la fase de análisis para mejorar el plugin una vez se empezó esta fase.

- Fase de documentación
  Habiendo realizado el desarrollo del plugin se realiza la fase de documentación, donde se exponen todas las fases anteriormente mencionadas haciendo hincapié además del funcionamiento de las tecnologías utilizadas y los protocolos que han formado parte de la implementación.

6.1.2. Diagrama de Gantt

La línea temporal que ha seguido el proyecto ha sido la siguiente:

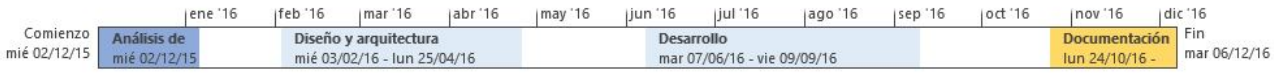

Figura 58: Escala temporal del proyecto

Se puede observar que entre las fases del proyecto ha habido tres períodos grandes de descanso, el primero de veinte días, el segundo y el tercero de treinta días cada uno. También ha habido un período corto de paro en una de las subtareas. Se puede entender mejor en la siguiente tabla que refleja las tareas y la duración de las mismas:

Nombre de tarea	Duración	Comienzo	Fin	Predecesoras	Nombres de los recursos
Análisis de requisitos	**25 días**	**mié 02/12/15**	**mar 05/01/16**		**Patricia Arias Cabarcos Florina Almenárez Mendoza**
Estudio del protocolo OpenID Connect	10 días	mié 02/12/15	mar 15/12/15		Rasha Aljelani Shellick
Análisis de analizadores de trazas	8 días	mié 16/12/15	vie 25/12/15	2	Rasha Aljelani Shellick Patricia Arias Cabarcos Florina Almenárez Mendoza
Definición de escenarios	5 días	lun 28/12/15	vie 01/01/16	3	Rasha Aljelani Shellick Patricia Arias Cabarcos Florina Almenárez Mendoza
Análisis final	2 días	lun 04/01/16	mar 05/01/16	3;2	Rasha Aljelani Shellick Patricia Arias Cabarcos Florina Almenárez Mendoza
Diseño y arquitectura	**59 días**	**mié 03/02/16**	**lun 25/04/16**		**Patricia Arias Cabarcos Florina Almenárez Mendoza**
Estudio de las tecnologías de autenticación	3 días	mié 03/02/16	vie 05/02/16	5FC+20 días	Rasha Aljelani Shellick
Despliegue del servidor y cliente OpenAM	30 días	lun 08/02/16	vie 18/03/16	7	Rasha Aljelani Shellick
Diseño de un disector en LUA	19 días	lun 28/03/16	jue 21/04/16	8FC+5 días	Rasha Aljelani Shellick
Diseño del plugin	2 días	vie 22/04/16	lun 25/04/16	9	Rasha Aljelani Shellick
Desarrollo	**69 días**	**mar 07/06/16**	**vie 09/09/16**		**Rasha Aljelani Shellick**
Desarrollo del disector	10 días	mar 07/06/16	lun 20/06/16	10FC+30 días	Rasha Aljelani Shellick
Desarrollo de los campos del disector	4 días	mar 21/06/16	vie 24/06/16	12	Rasha Aljelani Shellick
Integración de los datos mostrados	7 días	lun 27/06/16	mar 05/07/16	13	Rasha Aljelani Shellick
Integración de las funciones diferenciadoras de flujo	18 días	mié 06/07/16	vie 29/07/16	14	Rasha Aljelani Shellick
Integración de las funciones diferenciadoras de petición	14 días	lun 01/08/16	jue 18/08/16	15	Rasha Aljelani Shellick
Realización de pruebas	16 días	vie 19/08/16	vie 09/09/16	12;13;14;15;16	Rasha Aljelani Shellick
Documentación	**32 días**	**lun 24/10/16**	**mar 06/12/16**		**Patricia Arias Cabarcos Florina Almenárez Mendoza**
Documentación del proyecto	32 días	lun 24/10/16	mar 06/12/16	17FC+30 días	Rasha Aljelani Shellick

Figura 59: Tabla de tareas identificativas del proyecto

Para estimar la duración total del proyecto se realizará un diagrama de Gantt teniendo en cuenta tanto la duración del proyecto como de las pausas tomadas:

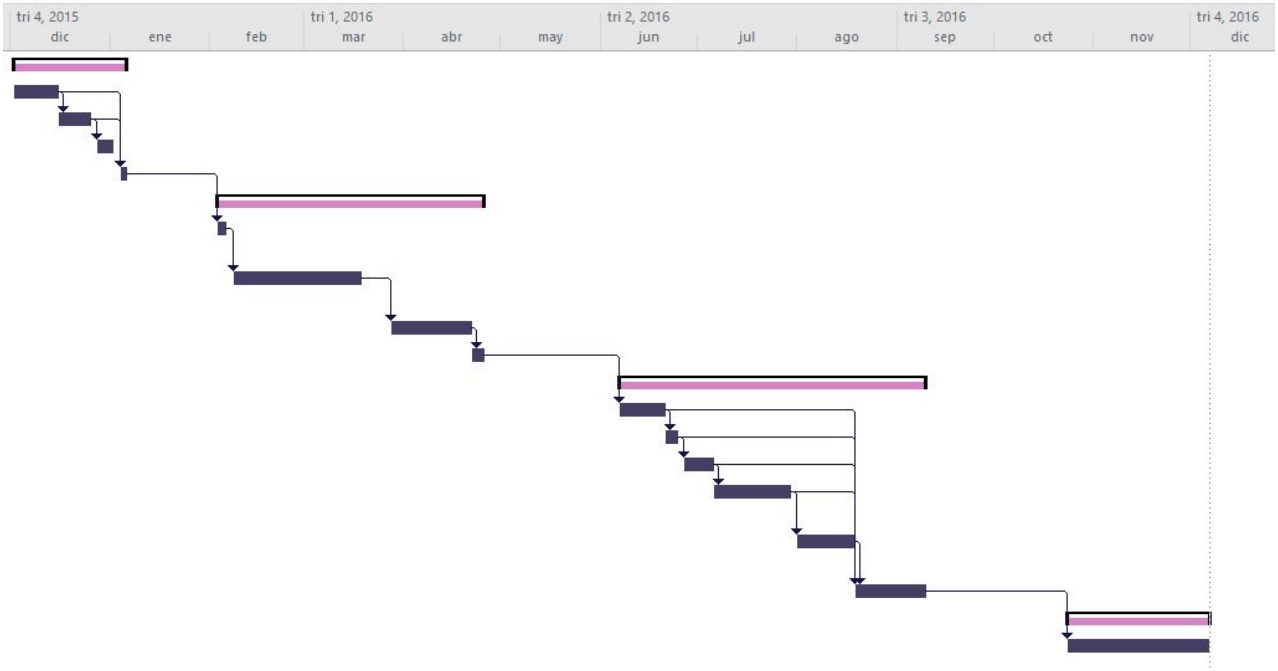

Figura 60: Gantt del proyecto

6.2. Presupuesto

Después de realizar la planificación se utilizan las horas empleadas y calculadas en el apartado anterior para estimar el coste de todo el proyecto.

6.2.1. Coste de personal

En este apartado se detalla el coste del personal que ha estado implicado en el proyecto.
Se ha tenido en cuenta, como se ha podido observar en el apartado **6.1. Planificación**, que se han necesitado tres personas para la realización del despliegue:

1. El segundo perfil es de ingeniero sin experiencia que ha llevado a cabo las tareas de desarrollo y despliegue de la infraestructura según los requisitos marcados.
   Se ha tenido en cuenta que el máximo anual para proyectos de investigación de la Universidad Carlos III es de 8,8 hombres/mes, lo que equivale a 1155 horas/anuales, pero al tener una persona que no ha estado implicada totalmente al proyecto se ha tenido en cuenta el máximo anual de dedicación genérico y es de 12 hombres/mes, es decir 1575 horas anuales. Por ello, teniendo en cuenta que por tener que compaginar el proyecto con otros temas se le ha dedicado un 70 % del tiempo, lo que equivaldría a 8,4 hombres/mes.

2. Se han implicado dos sujetos con perfiles ingeniero Senior para la gestión y el análisis del proyecto. Por ello han estado implicados más en la primera fase de análisis de requisitos pero se ha tenido en cuenta que han estado

supervisando cada una de las fases.
Han estado un total de 116 días en proyecto, pero 25 de ellos al 70 % teniendo en cuenta que se ha tenido que compaginar el proyecto con otros temas y los 91 días restantes han tenido un trabajo de supervisión, lo que en porcentajes de tiempo sería de 40 % en ese periodo. Es decir, por cada uno de los perfiles siguiendo el mismo razonamiento que el perfil más bajo, es de 3,5 hombres/mes.

Dado los anteriores puntos el coste ha sido el siguiente: Da un total de 52.659,66

Apellidos y nombre	**Categoría**	**Dedicación (Hombre mes)**	**Coste (hombre mes)**	**Coste total**
Arias Cabarcos, Patricia	Ingeniero Senior	3,5	4.289,54	15.013,39
Almenárez Mendoza, Florina	Ingeniero Senior	3,5	4.289,54	15.013,39
Aljelani Shellick, Rasha	Ingeniero	8,4	2.694,39	22.632,88

Tabla 20: Coste del personal en el proyecto

Euros en gastos.

6.2.2. Coste de equipos

El coste asociado a los recursos materiales empleado abarcaría desde el software hasta el hardware y se pueden ver desglosados en la siguiente tabla:

Descripción	**Coste (Euros)**	**% Uso dedicado al proyecto**	**Dedicación (meses)**	**Periodo de depreciación**	**Coste imputable**
Ultrabook Samsung serie 5	1100	100	12	48	275
Oneplus 3T	449	100	1	24	18,71
Microsoft Windows 10 Professional	251	100	6	12	125,5
Microsoft Office 365 Universitarios	79	100	6	12	39,5
Wireshark	0	100	12	12	0
Cliente OpenAM	0	100	12	12	0
Servidor OpenAM	0	100	12	12	0

Tabla 21: Coste del equipo en el proyecto

La fórmula de amortización utilizada es la siguiente:

$$\frac{A}{B} x C x D \tag{1}$$

Los parámetros que aparecen en ella se pueden definir de la siguiente manera:
A = meses desde la fecha de facturación de los equipos
B = periodo de depreciación
C = coste del equipo (sin IVA)
D = % del uso que se dedica al proyecto

Da un un total de 458,71 Euros de costes de equipo.

6.2.3. Otros costes directos

Existen también los costes indirectos que tienen que ser incluidos en el presupuesto del proyecto y por ello se ha tenido en cuenta los siguientes:

Descripción	**Coste imputable**
Internet	55
Eletricidad	60
Material fungible	200

Tabla 22: Otros costes directos en el proyecto

Este apartado incluye todos los gastos no contemplados en los conceptos anteriores, que en su total da 315 Euros.

6.2.4. Coste total

Teniendo en cuenta todos los gastos que se han tenido en cuenta en el apartado **6.2. Presupuesto** podemos englobarlos de la siguiente manera y calcular con ello el coste total del proyecto:

Presupuesto Costes Totales	**Presupuesto Costes Totales**
Personal	52.660
Amortización	459
Costes de funcionamiento	315
Costes Indirectos	4.681
Total	64.120

Tabla 23: Costes totales del proyecto

El presupuesto de este proyecto asciende a un total de **SESENTA Y CUATRO MIL CIENTO VEINTE** Euros, incluyendo el iva que va incluido en los costes indirectos.

7. Conclusiones y líneas futuras

7.1. Conclusiones

Esta era donde la tecnología se ha vuelto un pilar importante de nuestro día a día y donde el número de aplicaciones va en aumento de forma exponencial, es importante conocer los métodos que se utilizan para securizar las comunicaciones y la información privada que tenemos en la red.

No estamos hablando únicamente de los protocolos de seguridad en la transmisión y el intercambio de datos, sino que debido a que utilizamos más aplicaciones en las cuales tenemos información relevante sobre nosotros mismos, trabajos o gustos personales, estamos rebajando el nivel de seguridad que utilizamos a la hora de elegir la contraseña o el usuario recayendo en el uso de la repetición de estas mismas o en peores casos, el anotarlas en cualquier lugar poco seguro.

Esta es una de las principales razones de la existencia de los protocolos de autenticación, ayudarnos a tener cuentas de usuario que centralizan y son utilizadas por varias aplicaciones a la vez evitando esta redundancia tan peligrosa en la época. Pero tampoco podemos confiarnos en la utilización de estos protocolos sin conocerlos en profundidad y por ello, tras la larga investigación que se ha realizado para documentar este proyecto y para la creación del plugin, podemos garantizar que este tipo de protocolos intenta garantizar esta seguridad con los máximos mecanismos posibles para comprobar que los datos no han sido modificados en el proceso mismo de autenticación.

Además, dada la gran importancia del tipo de protocolos analizados en este documento, llegamos a la conclusión de que es trascendental el conocimiento de su funcionamiento al igual que el conocimiento de su implementación, y por esta razón se ha creado un plugin al cual se ha enfocado para darle un toque didáctico para la comprensión de las trazas intercambiadas y enviadas por cada una de las partes implicadas, es decir, servidor, cliente y usuario final, tanto a la parte mostrada como al mismo código de programación, donde explica los principales puntos el protocolo.

Cabe destacar que el estudio para la realización del proyecto que ha sido realizado en distintas web y foros, tanto en las páginas de especificaciones del propio protocolo, OpenID Connect, en foros donde se argumentaba las mejores formas de programación de plugins en Wireshark y en la web del mismo proveedor, no se hubiera podido llevar a cabo si no fuera porque todas las aplicaciones utilizadas en este proyecto fueron de carácter gratuito, y gracias al material facilitado ayudaron a la hora de la comprensión de cada una de las partes implicadas.

Es cierto que las decisiones tomadas a la hora de la implementación de la infraestructura y del desarrollo del plugin ha sido personal y podría realizarse de multitud de formas, pero consideramos que gracias a la programación del plugin de la forma que se ha realizado, haciendo un estudio traza por traza intercambiada entre el servidor y el cliente, y comentando en el código de LUA un resumen de cada uno de los flujos y las peticiones que se intercambian, al igual que mostrando

los parámetros más importante de cada uno de ellos en la interfaz de Wireshark, se ha creado una herramienta que ayuda al entendimiento del protocolo OpenID Connect.

En conclusión, consideramos que este documento ha cumplido los objetivos establecidos por el proyecto, aportando una visión objetiva y detallada de la infraestructura utilizada y del desarrollo del mismo plugin, al igual que de las decisiones tomadas a la hora de la elección de las distintas partes.

7.2. Líneas futuras

El desarrollo del plugin se ha llevado de la forma más didáctica posible en una infraestructura de pruebas y donde tenemos el control de ambas partes, por lo que no ha habido necesidad de tomar medidas extras para la seguridad. Igualmente, para tener un enfoque más objetivo y real a la hora de la realización de un plugin para el estudio del protocolo OpenID Connect, sería ideal tomar las siguientes medidas:

- En la infraestructura creada se podría securizar debidamente las trazas utilizando TLS para el intercambio de información entre cliente y servidor. Es cierto que no se ha podido realizar este proceso debido a que la información analizada por parte del cliente no se podría haber llevado a cabo activando SSL y utilizando un certificado en el proceso, pero si escogiéramos un Cliente más completo para la autenticación podríamos activar esta opción y tener un escenario más real.

- Se podría implementar el plugin en Wireshark con el objetivo de tener una herramienta más óptima en vez de más didáctica, para que muestre los resultados más rápido y tenga cierta latencia como la tiene el plugin en capturas de trazas muy grandes.

- Otra opción interesante sería la realización de este mismo plugin pero directamente en navegador, para observar las trazas enviadas del cliente al servidor a tiempo real y en una misma interfaz.

- Utilizando un entorno real a la hora de la implementación del plugin, tendríamos resultados más auténticos y mucho más personalizados. Podríamos configurar la herramienta para que muestre resultados más concisos y auténticos específicos de un entorno empresarial.

- El cliente utilizado para el desarrollo de este plugin no contenía un apartado de Flujo Híbrido para realizar pruebas de este tipo. Por ello, como medida para tomar en un futuro, sería interesante realizar un despliegue con un cliente que si contenga este tipo de autenticación para poder desarrollar el único flujo restante y conseguir así un plugin más completo.

8. Bibliografía

La bibliografía consultada es la siguiente:

[1] **Gestión de identidades, ID management**. Disponible en http://searchdatacenter.techtarget.com/es/definicion/Gestion-de-identidades-ID-management . Fecha de última consulta: junio 2017

[2] **DRAFT NIST Special Publication 800-63B Digital Identity Guidelines**. Disponible en https://pages.nist.gov/800-63-3/sp800-63b.html . Fecha de última consulta: junio 2017

[3] Kim Tracy. **Ïdentity management systems"**, IEEE, páginas 34 - 37, 31 de octubre 2008.

[4] **OpenID Connect Core 1.0 incorporating errata set 1**. Disponible en http://openid.net/specs/openid-connect-core-1_0.html . Fecha de última consulta: junio 2017

[5] **OAuth 2.0 Multiple Response Type Encoding Practices**. Disponible en http://openid.net/specs/oauth-v2-multiple-response-types-1_0.html . Fecha de última consulta: junio 2017

[6] **The OAuth 2.0 Authorization Framework**. Disponible en https://tools.ietf.org/html/rfc6749 . Fecha de última consulta: junio 2017

[7] Kuinam J. KimCheong Ghil KimTaeg Keun WhangboKyoungro Yoon. **.ª continuous playing scheme on RESTful web service"**, Cluster Comput (2016) 19: 379, 18 January 2016.

[8] **Security Assertion Markup Language (SAML) 2.0 Profile for OAuth 2.0 Client Authentication and Authorization Grants**. Disponible en https://tools.ietf.org/html/rfc7522 . Fecha de última consulta: junio 2017

[9] **How to Study and Learn SAML**. Disponible en http://identitymeme.org/doc/draft-hodges-learning-saml-00.html . Fecha de última consulta: junio 2017

[10] **Introducción a JSON**. Disponible en http://www.json.org/json-es.html . Fecha de última consulta: junio 2017

[11] **JSON Web Token (JWT)**. Disponible en http://self-issued.info/docs/draft-ietf-oauth-json-web-token.html . Fecha de última consulta: junio 2017

[12] **Certified OpenID Connect Implementations**. Disponible en http://openid.net/developers/certified/ . Fecha de última consulta: junio 2017

[13] **Página oficial de Wireshark**. Disponible en https://www.Wireshark.org/ . Fecha de última consulta: junio 2017

[14] **Manual de Referencia de Lua 5.1**. Disponible en https://www.lua.org/manual/5.1/es/ . Fecha de última consulta: junio 2017

[15] **SAML DevTools extension**. Disponible en https://chrome.google.com/webstore/detail/saml-devtools-extension/jndllhgbinhiiddokbeoeepbppd . Fecha de última consulta: junio 2017

[16] **Wireshark Developer's Guiden**. Disponible en https://www.Wireshark.org/docs/wsdg_html_chunked/ . Fecha de última consulta: junio 2017

[17] **OpenAM Administration Guide**. Disponible en

https://backstage.forgerock.com/docs/openam/13.5/admin-guide/chap-openid-connect . Fecha de última consulta: junio 2017

9. Glosario

- **Issuer Identifier:**
  Identificador Verificable para un Emisor. Es una dirección URL sensible a mayúsculas y minúsculas utilizando el esquema https que contiene esquema, host y opcionalmente, número de puerto y componentes de ruta y componentes de consulta o fragmento.

- **Lenguajes scripting:**
  Son un tipo de lenguaje de programación interpretado, es decir, existe otro programa que procesa los comandos y los ejecuta. Algunos ejemplos de este tipo son Python, Javascript, Ruby, etc.

- **Hash:**
  Función resumen que se utiliza para crear un resumen de toda la información que se le ha introducido como entrada. Sirve para comprobar que los datos han sido comprometidos, o no, en alguna parte del proceso.

- **Cifrar:**
  Es el método que permite volver ilegible una cadena de caracteres aplicando un algoritmo especial de modificación mediante el uso de una clave secreta.

- **Codificar:**
  Es el método que permite una cadena de caracteres en un determinado lenguaje transformarla en otra distinta utilizando otro. Es decir, altera la semántica de la misma cadena de caracteres.

- **Codificación de consultas:**
  En este modo, los parámetros de respuesta de autorización se codifican en la cadena de consulta agregada al *redirect_uri* al redirigir de nuevo al cliente.

- **Codificación de fragmentos:**
  En este modo, los parámetros de respuesta de autorización se codifican en el fragmento añadido al *redirect_uri* al redirigir de nuevo al cliente.

- **Userdata**
  LUA proporciona un tipo básico para representar los valores de arrays ofreciéndonos un área de memoria sin operaciones predefinidas con lo que se puede gestionar el contenido sin ninguna restricción de forma.

- **Schema**
  Estructura que se le asigna al Uniform Resource Identifier (URI) y que se organiza de la siguiente manera:
  [//[usuario[:contraseña]@]host[puerto]][/path][?query][#fragmento]

- **CSRF**
  Acrónimo de Cross-site request forgery, es un tipo de programa malicioso de un sitio web que tiene como objetivo la falsificación de petición en sitios que intercambian información mutuamente y haciéndose pasar por el otro. Esta vulnerabilidad es también conocida como XSRF.

- **XHTML**
  Acrónimo de eXtensible HyperText Markup Language, que es el lenguaje en HTML cambiado a formato XML para su tratamiento.

10. Anexo

10.1. Instalación del Servidor OpenAM

Para la instalación el sevidor OpenAM debemos seguir los siguientes pasos:

1. Lo primero sería descargar el fichero del Servidor OpenAM de la web del fabricante, y siempre la versión Nightly que es la versión gratuita.
   Una vez se tenga, hay que descomprimir el contenido del fichero y centrarse en un fichero *.war* que contiene el servidor:

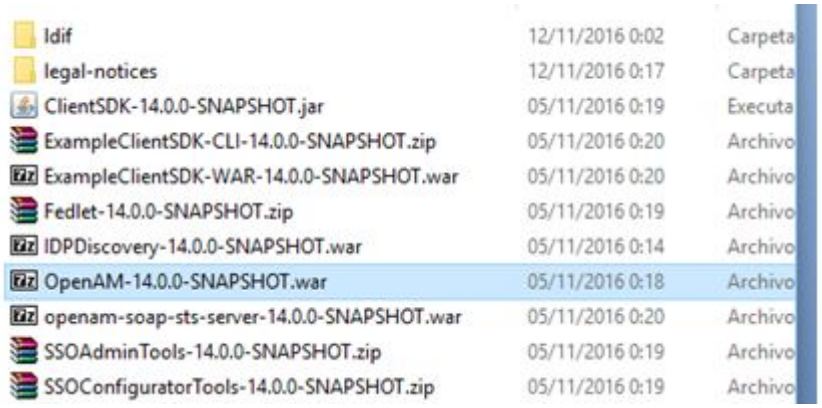

Figura 61: Fichero Nightly OpenAM

2. Ese fichero se guarda en el servidor Web, que en este caso ha sido un Tomcat:

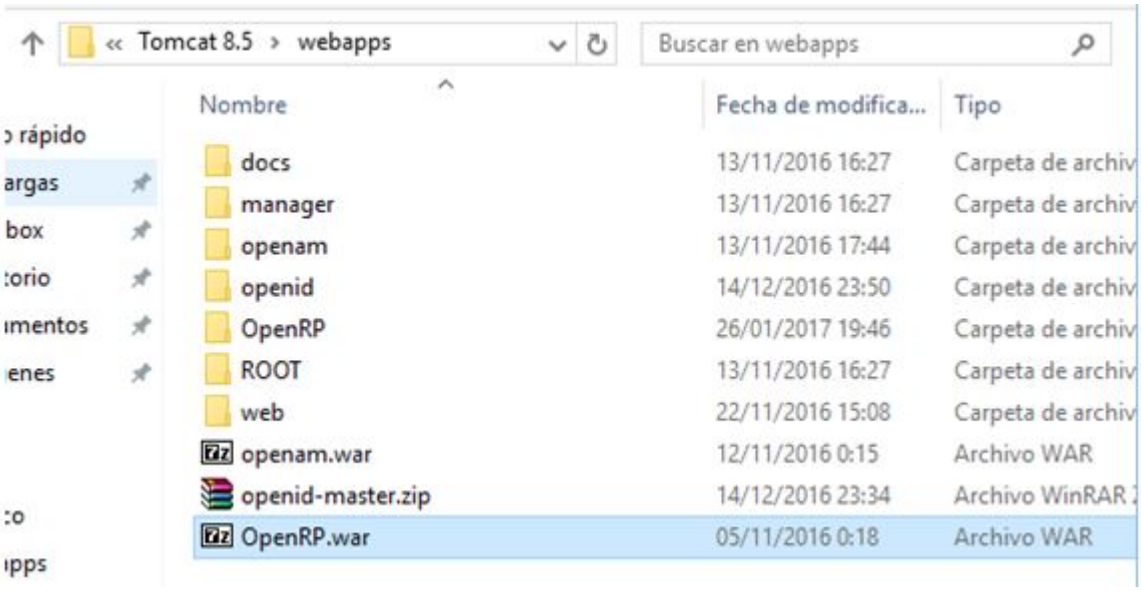

Figura 62: Carpeta de Tomcat con el fichero OpenAM

3. Una vez puesto el fichero OpenAM en la carpeta se ejecuta Tomcat y se accede a la url http://127.0.0.1:puerto/OpenRP:

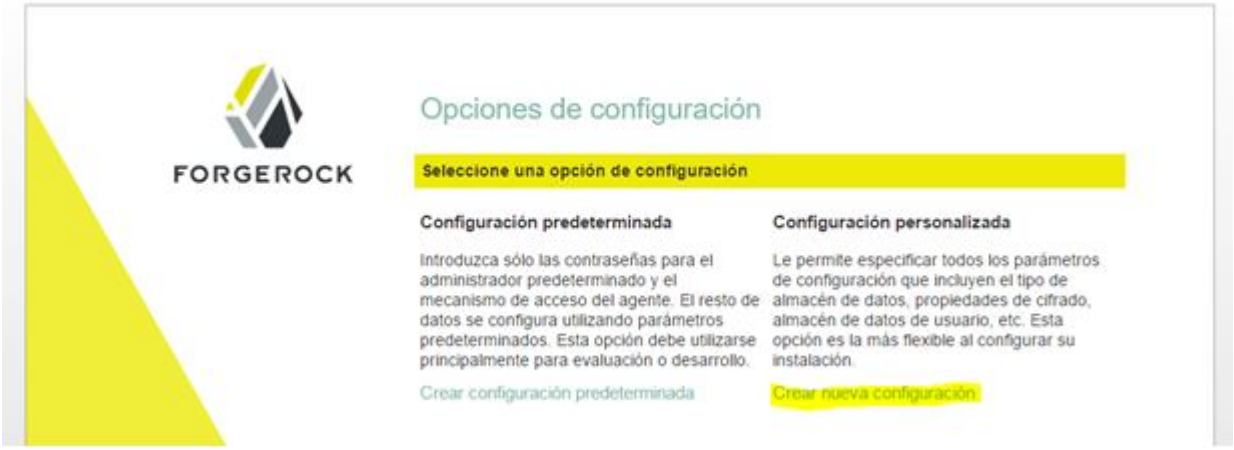

Figura 63: Menú principal del servidor OpenAM

4. Pulsamos en la opción remarcada en la **Figura 63**, *Crear nueva configuración*, y aparecerá la siguiente ventana para aceptar la licencia del producto:

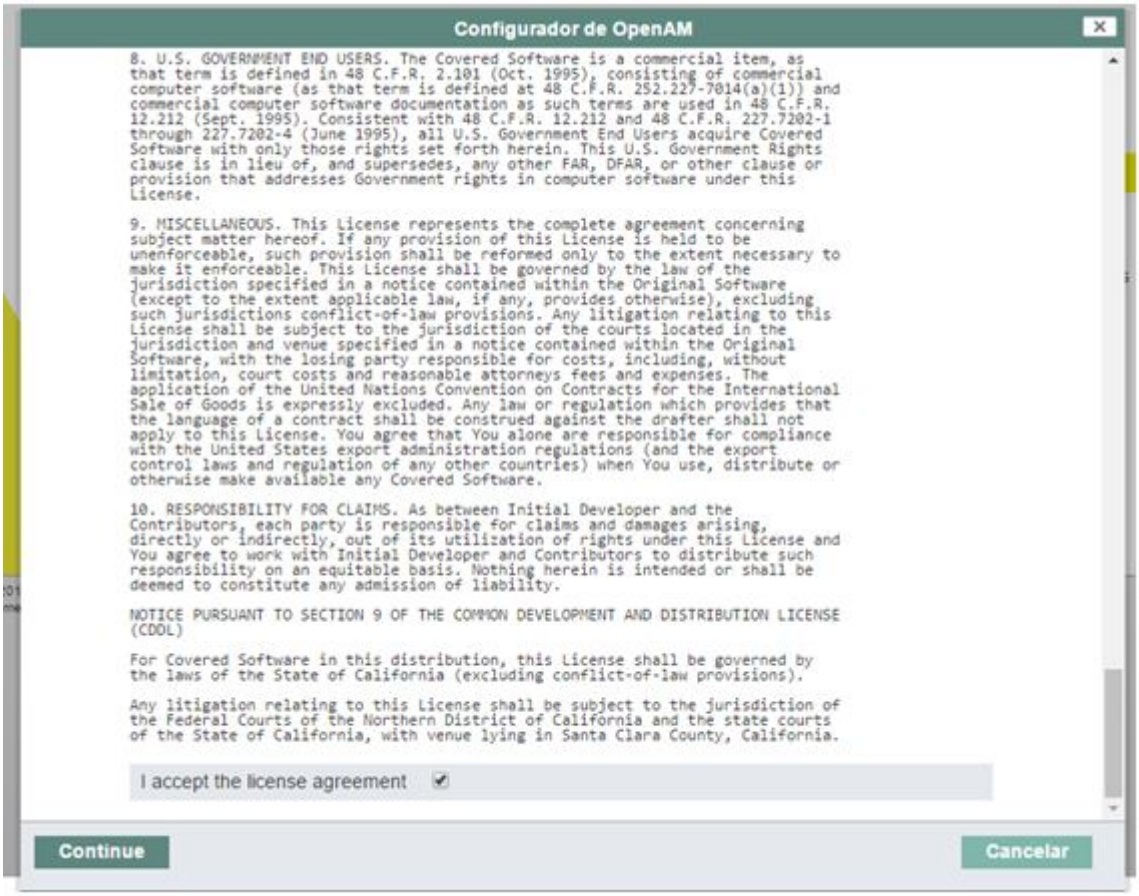

Figura 64: Licencia del servidor OpenAM

5. Una vez seleccionado, el siguiente menú desplegable dará la opción de elegir

la contraseña del usuario administrador, *amAdmin*:

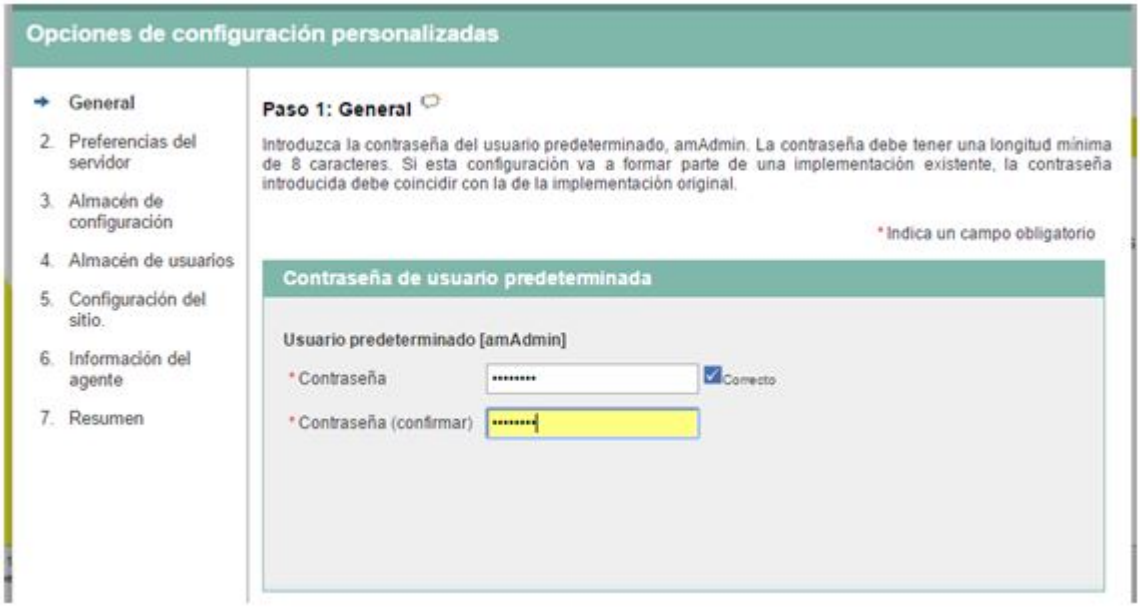

Figura 65: Configuración del administrador OpenAM

6. Seleccionar una contraseña y presionar en siguiente. Una vez pulsado se muestra la ventana que se observa en la *Figura 66*:

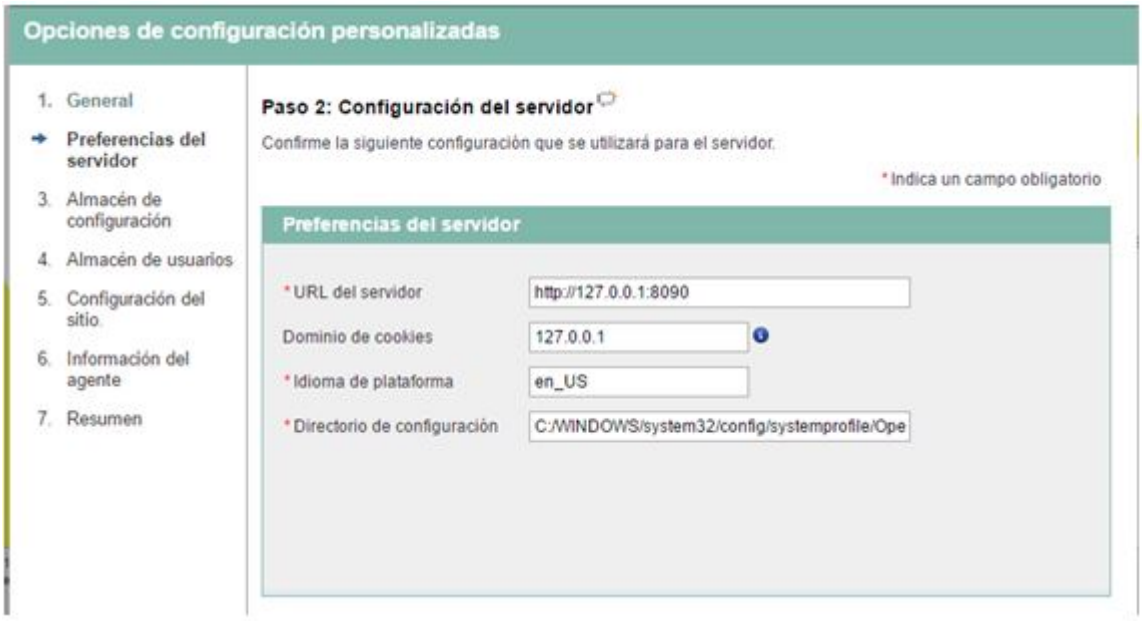

Figura 66: Configuración por defecto del servidor OpenAM

7. Modificar la Url del servidor y el dominio de cookies por el del host o la IP pública que se tenga:

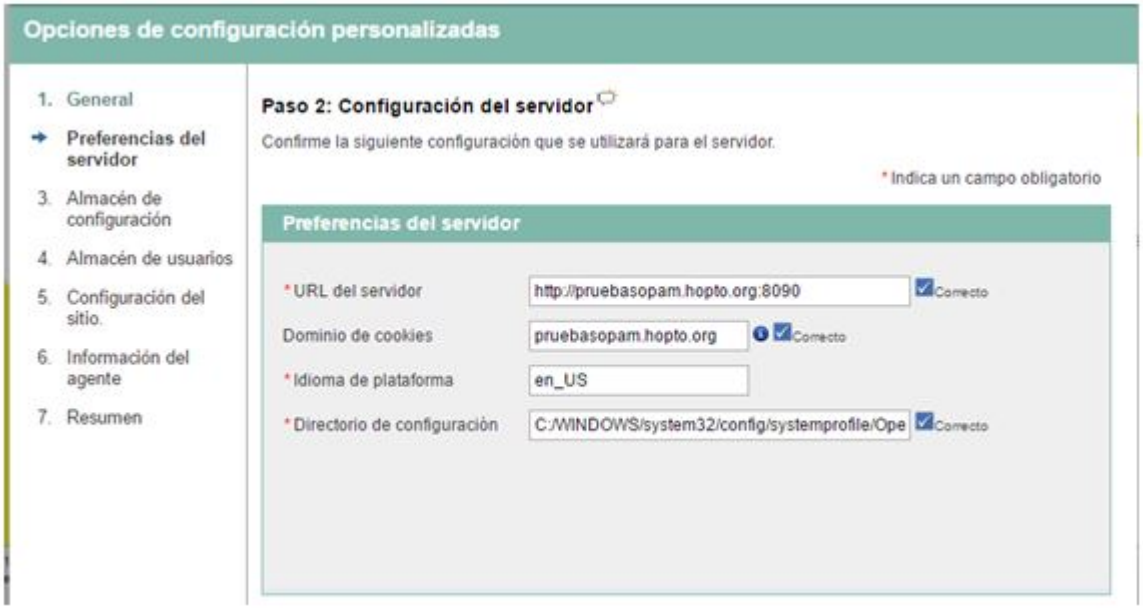

Figura 67: Configuración del servidor OpenAM

8. Una vez configurado, presionar en siguiente y se muestra la ventana de configuración de puertos:

Figura 68: Configuración del servidor OpenAM 2

9. Escoger la opción de *primera instancia* y se continua. En la siguiente ventana

se cambia la configuración por defecto por la opción de *Almacén de datos de usuario de OpenAM*:

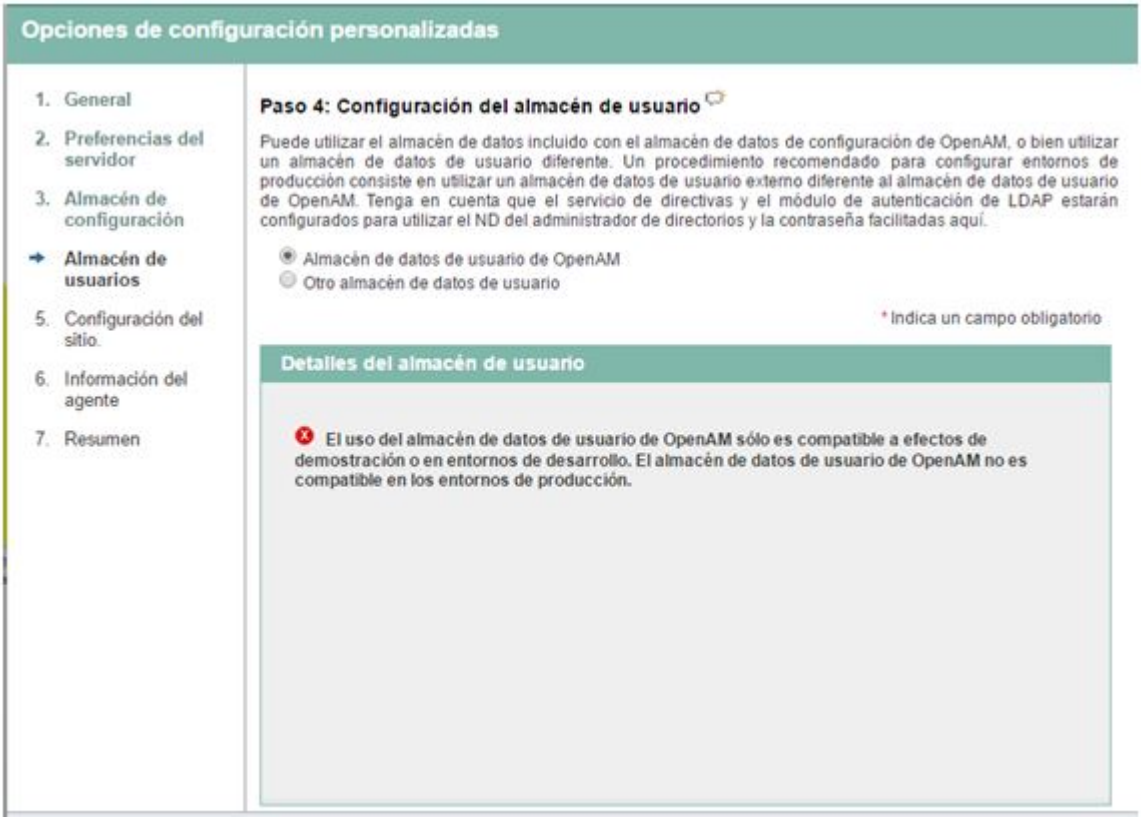

Figura 69: Almacenamiento del servidor OpenAM

10. Presionar en siguiente sin modificar nada más y aparece una ventana que muestra la *configuración de sitio* que se dejará igual a como aparece en la *Figura 70*:

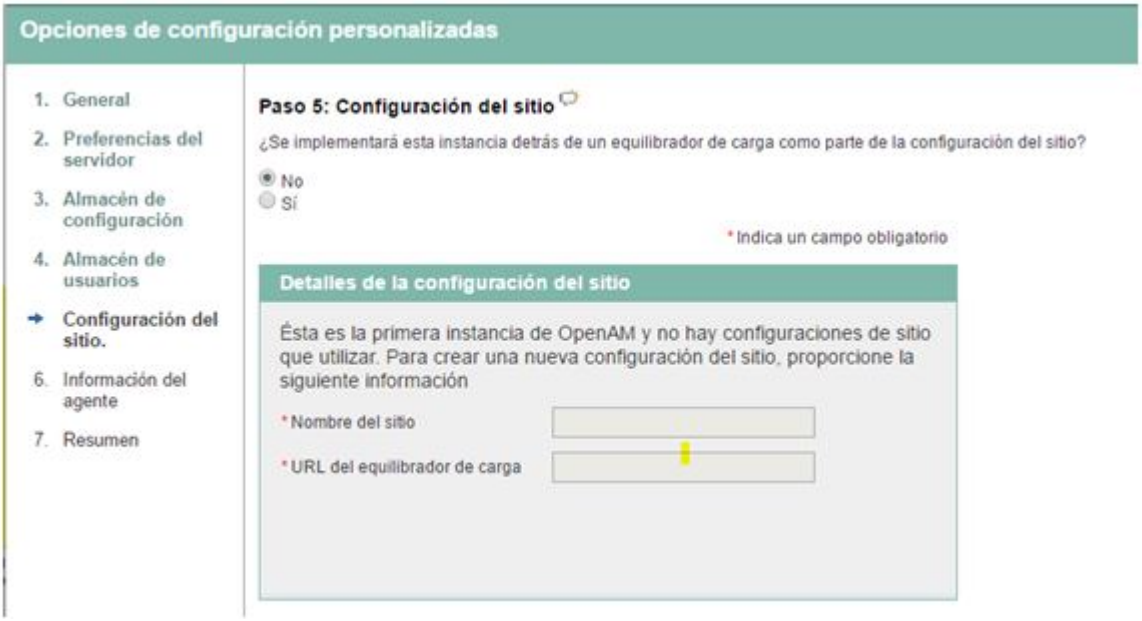

Figura 70: Implementación de un equilibrador de carga

11. En la siguiente ventana se pide una contraseña para el agente de directivas predeterminado, *UrlAccessAgent* por lo que se escoge una que no coincida con la del usuario amAdmin:

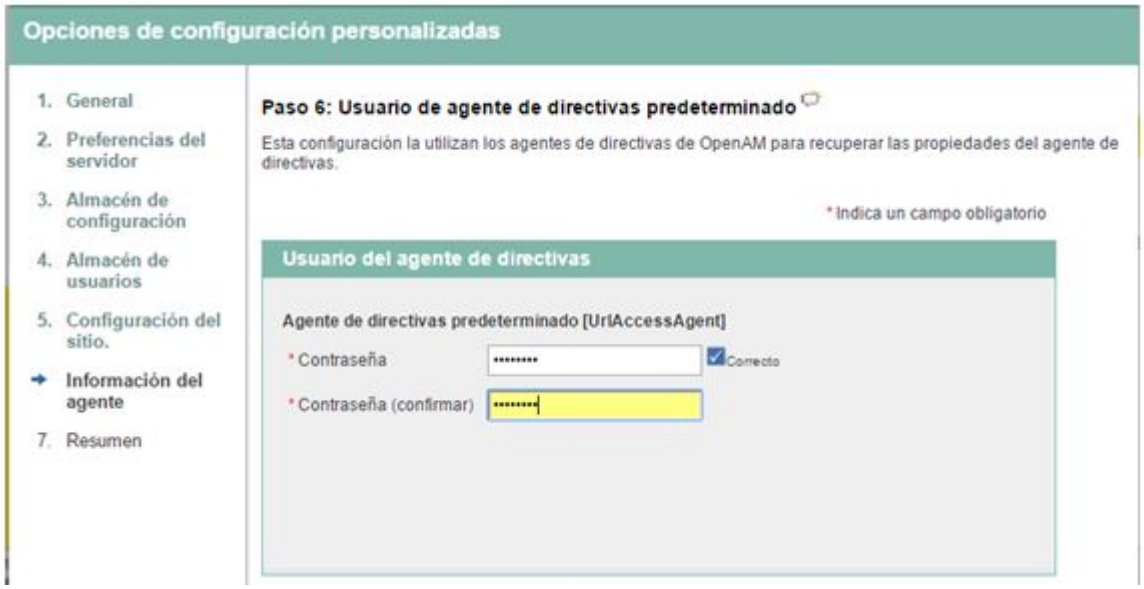

Figura 71: Configuración del agente de directivas

12. Con estos pasos se termina la configuración de OpenAM y aparece la ventana donde nos muestra el resumen de la configuración que se puede observar en la *Figura 72*:

Figura 72: Resumen de configuración del servidor OpenAM

13. Finalmente le damos a crear configuración y se espera a que se realice la instalación de OpenAM:

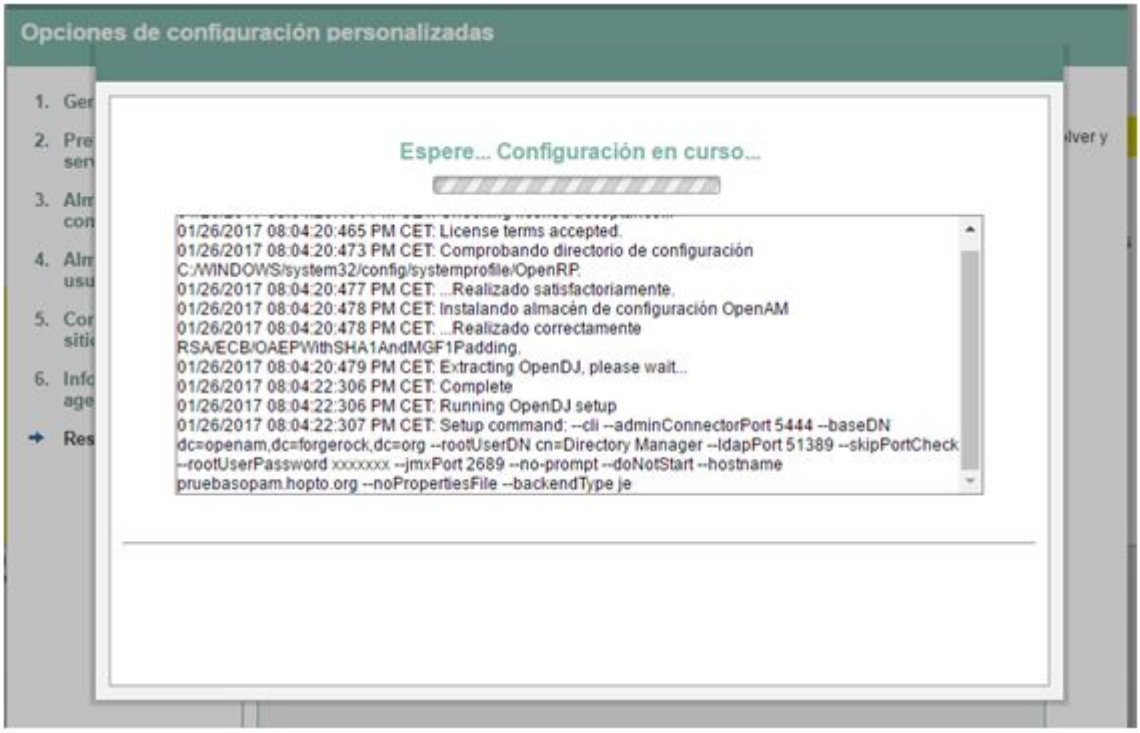

Figura 73: Instalación del servidor OpenAM

14. Una vez instalado el servidor se configura para que utilice el protocolo OpenID Connect para la autenticación. Para ello hay que escoger el *realm* y una vez dentro escogemos la opción de *Configure Oauth Provider*. Después la de OpenID Connect para poder configurarlo de la manera que se muestra en la *Figura 74*, que es la de por defecto:

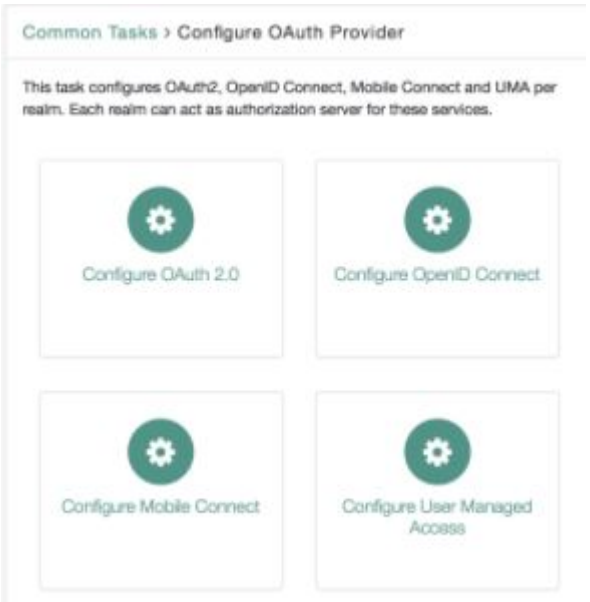

Figura 74: Configuración del protocolo OpenID Connect

15. Una vez se tenga hay que configurar un agente con los datos del RP tal como se explica en el apartado **4.2. Cliente OpenAM** y con ello ya está listo el servidor OpenAM.